Docket: 43876-156 **PATENT**

IN THE UNITED STATES PATENT AND TRADEMARK OFFICE

In re Application of	:	Customer Number: 20277
HANSEN et al.	:	Confirmation Number: 5955
Application No.: 10/757,866	:	Group Art Unit: 2183
Filed: January 16, 2004	:	Examiner: Eric Coleman

For: METHOD AND SOFTWARE FOR STORE MULTIPLEX OPERATION

SUPPLEMENTAL DECLARATION OF KORBIN VAN DYKE

I, Korbin Van Dyke, state that:

Summary of My Opinions

1. I previously submitted a declaration in connection with this proceeding (see Van Dyke Declaration dated August 15, 2007), referenced hereinafter as the "initial Van Dyke declaration". For brevity, I will not repeat information set forth in the initial Van Dyke declaration in this declaration.

2. In preparation of this declaration I have reviewed U.S. Patent Application Serial No. 10/757,866. I have also reviewed U.S. Patent Nos. 5,742,840 and 6,295,599 (respectively the '840 and '599 patents) that the 10/757,866 patent application indirectly claims priority to, as well as appendices to the '840 and '599 patents (the Terpsichore and Zeus System Architecture manuals, respectively, and hereinafter referred to respectively as the Terpsichore and the Zeus manuals). I have reviewed the Office Action for the 10/757,866 patent application mailed on November 5, 2007, including the paragraph on page 10 that discusses the Response to Arguments and particularly the Examiner's conclusion that the priority for the claimed invention does not extend to the '840 or the '599 patents, since limitations of claims 9, 18, 33, 34, 40, and 41 are not supported by the '840 or the '599 patents. My understanding is that the features of the claimed invention are taught and supported by complying with the written description requirement and the enablement requirement. My understanding of the written description requirement is that a patent disclosure must describe the claimed invention in sufficient detail that one of ordinary skill in the art can reasonably conclude that the inventor had possession of

the claimed invention at the time of filing the patent disclosure. My understanding of the enablement requirement is that the patent disclosure must contain sufficient information regarding the subject matter of the claims to enable one of ordinary skill in the pertinent art to make and use the claimed invention. I further understand that whether the enablement requirement is met depends on whether undue experimentation is necessary for one of skill in the art to practice the invention in light of the patent disclosure.

3. Based on my review of the materials identified in paragraph 2 of this declaration, it is my opinion that with respect to the following limitations relating to claims 9, 18, 33, 34, 40, and 41 (as amended), the disclosures of the '840 patent and the '599 patent each indicate that the inventors were in possession of the claimed invention of the 10/757,866 patent application as of the August 16, 1995 filing date of the '840 patent and further as of the August 24, 1999 filing date of the '599 patent; and further the disclosures of the '840 patent and the '599 patent each would have enabled a person of ordinary skill in the art to make and use, without undue experimentation, the claimed invention of the 10/757,866 patent application as of the August 16, 1995 filing date of the '840 patent, and further as of the August 24, 1999 filing date of the '599 patent. The limitations referred to are:

{claim 9} "decoding a second single instruction specifying a register containing a first plurality of floating-point operands and another register containing a second plurality of floating-point operands"

"multiplying the first plurality of floating-point operands by the second plurality of floating-point operands to produce a plurality of products"

"providing the plurality of products to partitioned fields of a result register as a catenated result"

{claim 18} "wherein at least some of the instructions further include a group floating-point multiply instruction for multiplying floating-point data in the programmable processor, the group floating-point multiply instruction capable of instructing the programmable processor to perform operations"

the operations comprising "decoding the group floating-point multiply instruction specifying a register containing a first plurality of floating-point operands and another register containing a second plurality of floating-point operands"

the operations comprising "multiplying the first plurality of floating-point operands by the second plurality of floating-point operands to produce a plurality of products"

the operations comprising "providing the plurality of products to partitioned fields of a result register as a catenated result"

{claim 33} "wherein each of the first and second operands has a width of 64 bits"

{claim 34} "a step of executing a plurality of different group floating-point arithmetic operations that arithmetically operate on multiple floating-point operands stored in partitioned fields of registers in the register file to produce a catenated result that is returned to a register in the register file, wherein the catenated result comprises a plurality of individual floating-point results"

{claim 40} "wherein each of the first and second operands has a width of 64 bits"

{claim 41} "wherein the plurality of instructions further comprises a plurality of different group floating-point arithmetic operations that arithmetically operate on multiple floating-point operands stored in partitioned fields of registers in the register file to produce a catenated result that is returned to a register in the register file, wherein the catenated result comprises a plurality of individual floating-point results"

Summary of '840 Analysis:

4. The disclosure of the '840 patent provides detailed information and description that I believe indicates that the inventors were in possession of the aforementioned limitations of claim 9 (as amended) of the 10/757,866 patent application, and that I further believe would have enabled a person of ordinary skill in the art to make and use the claimed invention without undue experimentation. For example, on at least pages 19-21 (describing floating-point data formats), 24-25 (describing general registers), 29 and 47-48 (describing floating-point arithmetic hardware), and 129-131 of the Terpsichore manual (describing details of Group Floating-point instructions such as various forms of Group Floating-point Multiply instructions) there are detailed descriptions of the aforementioned claim elements.

5. The aforementioned limitations of claim 18 (a computer-readable storage medium claim) are substantially similar to the aforementioned limitations of claim 9 (a method claim). Further, parent claim context providing antecedent basis for claim 18 (specifically "the execution unit") is substantially similar to corresponding parent claim context of claim 9. Thus, for at least the reasons described in paragraph 4 of this declaration, the disclosure of the '840 patent provides detailed information and description that I believe indicates that the inventors were in possession of the aforementioned limitations of claim 18 (as amended) of the 10/757,866 patent application, and that I further believe would have enabled a person of ordinary skill in the art to make and use the claimed invention without undue experimentation.

6. The disclosure of the '840 patent provides detailed information and description that I believe indicates that the inventors were in possession of the aforementioned limitations of claim 33 (as amended) of the 10/757,866 patent application, and that I further believe would have enabled a person of ordinary skill in the art to make and use the claimed invention without undue experimentation. For example, on at least pages 24-25 (describing general registers), 26 (generally describing store instructions), and 150-157 of the Terpsichore manual (describing details of Store and Store Immediate instructions such as various forms of Store Immediate and Store Multiplex Immediate instructions) there are detailed descriptions of the aforementioned claim elements.

7. The disclosure of the '840 patent provides detailed information and description that I believe indicates that the inventors were in possession of the aforementioned limitations of claim 34 (as amended) of the 10/757,866 patent application, and that I further believe would have enabled a person of ordinary skill in the art to make and use the claimed invention without undue experimentation. For example, on at least pages 19-21 (describing floating-point data formats), 24-25 (describing general registers), 29 and 47-48 (describing floating-point arithmetic hardware), and 129-131 of the Terpsichore manual (describing details of Group Floating-point instructions such as various Group Floating-point Add, Divide, and Multiply forms) there are detailed descriptions of the aforementioned claim elements.

8. The aforementioned limitations of claim 40 (a computer-readable storage medium claim) are substantially similar to the aforementioned limitations of claim 33 (a method claim). Further, parent claim context providing antecedent basis for claim 40 (specifically "the first and second operands") is substantially similar to corresponding parent claim context of claim 33. Thus, for at least the reasons described in paragraph 6 of this declaration, the disclosure of the '840 patent provides detailed information and description that I believe indicates that the inventors were in possession of the aforementioned limitations of claim 40 (as amended) of the 10/757,866 patent application, and that I further believe would have enabled a person of ordinary skill in the art to make and use the claimed invention without undue experimentation.

9. The aforementioned limitations of claim 41 (a computer-readable storage medium claim) are substantially similar to the aforementioned limitations of claim 34 (a method claim). Further, parent claim context providing antecedent basis for claim 41 (specifically "the execution unit") is substantially similar to corresponding parent claim context of claim 34. Thus, for at least the reasons described in paragraph 7 of this declaration, the disclosure of the '840 patent provides detailed information and description that I believe indicates that the inventors were in possession of the aforementioned limitations of claim 41 (as amended) of the 10/757,866 patent application, and that I further believe would have enabled a person of ordinary skill in the art to make and use the claimed invention without undue experimentation.

Summary of '599 Analysis:

10. The disclosure of the '599 patent provides detailed information and description that I believe indicates that the inventors were in possession of the aforementioned limitations of claim 9 (as amended) of the 10/757,866 patent application, and that I further believe would have enabled a person of ordinary skill in the art to make and use the claimed invention without undue experimentation. For example, on at least pages 14-16 (describing floating-point data formats), 19-20 (describing general registers), 23-24 and 55 (describing floating-point arithmetic hardware), and 258-260 of the Zeus manual (describing details of Ensemble Floating-point instructions such as various forms of Ensemble Multiply Floating-point instructions) there are detailed descriptions of the aforementioned claim elements. Note that in the Zeus manual, group floating-point instructions are termed "ensemble" floating-point instructions.

11. The aforementioned limitations of claim 18 are substantially similar to the aforementioned limitations of claim 9. Thus, for at least the reasons described in paragraph 10 of this declaration, the disclosure of the '599 patent provides detailed information and description that I believe indicates that the inventors were in possession of the aforementioned limitations of

claim 18 (as amended) of the 10/757,866 patent application, and that I further believe would have enabled a person of ordinary skill in the art to make and use the claimed invention without undue experimentation.

12. The disclosure of the '599 patent provides detailed information and description that I believe indicates that the inventors were in possession of the aforementioned limitations of claim 33 (as amended) of the 10/757,866 patent application, and that I further believe would have enabled a person of ordinary skill in the art to make and use the claimed invention without undue experimentation. For example, on at least pages 19-20 (describing general registers), 21 (generally describing store instructions), 123-125, and 128-130 of the Zeus manual (describing details of Store and Store Immediate instructions, including Store Multiplex and Store Multiplex Immediate forms) there are detailed descriptions of the aforementioned claim elements.

13. The disclosure of the '599 patent provides detailed information and description that I believe indicates that the inventors were in possession of the aforementioned limitations of claim 34 (as amended) of the 10/757,866 patent application, and that I further believe would have enabled a person of ordinary skill in the art to make and use the claimed invention without undue experimentation. For example, on at least pages 14-16 (describing floating-point data formats), 19-20 (describing general registers), 23-24 and 55 (describing floating-point arithmetic hardware), and 258-260 of the Zeus manual (describing details of Ensemble Floating-point instructions, such as various Ensemble Multiply, Add, and Divide forms) there are detailed descriptions of the aforementioned claim elements.

14. The aforementioned limitations of claim 40 are substantially similar to the aforementioned limitations of claim 33. Thus, for at least the reasons described in paragraph 12 of this declaration, the disclosure of the '599 patent provides detailed information and description that I believe indicates that the inventors were in possession of the aforementioned limitations of claim 40 (as amended) of the 10/757,866 patent application, and that I further believe would have enabled a person of ordinary skill in the art to make and use the claimed invention without undue experimentation.

15. The aforementioned limitations of claim 41 are substantially similar to the aforementioned limitations of claim 34. Thus, for at least the reasons described in paragraph 13 of this declaration, the disclosure of the '599 patent provides detailed information and description that I believe indicates that the inventors were in possession of the aforementioned limitations of claim 41 (as amended) of the 10/757,866 patent application, and that I further believe would have enabled a person of ordinary skill in the art to make and use the claimed invention without undue experimentation

16. A detailed explanation of the basis for my opinions is set forth in the remainder of this declaration.

Detailed Basis for My Opinions

Analysis of the disclosures of the '840 and the '599 patents:

17. For brevity, the following analysis focuses on and provides details relating to the '840 patent, while reciting summary information pointing out where similar descriptive information is provided in the '599 patent.

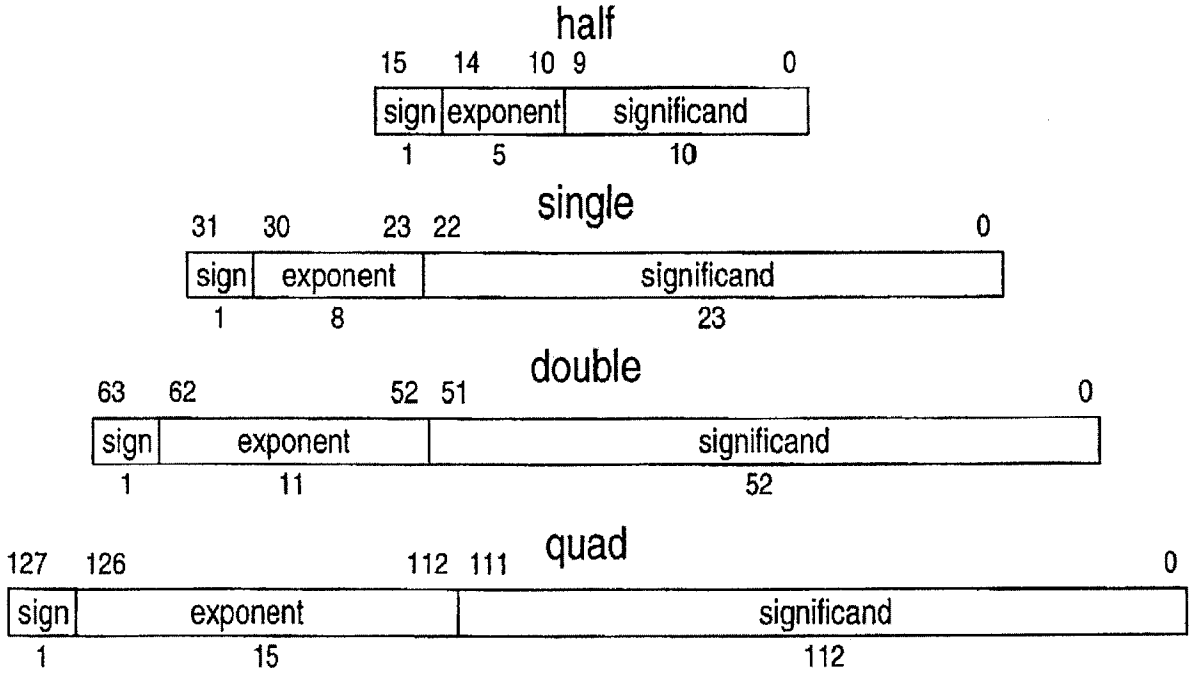
18. As discussed in paragraph 20 of the initial Van Dyke declaration, the '840 patent describes structure of a general purpose, programmable media processor (including, for example, a register file and an execution unit), and the '840 patent recites that an instruction set for the general purpose media processor is described by the Microfiche Appendix (referred to herein as the Terpsichore manual). In addition to elements discussed in paragraph 20 of the initial Van Dyke declaration, the '840 patent also describes that a unified stream of media data is processed by storage into the register file 110, and multi-precision arithmetic operations are performed on the media data. The operations include Boolean, integer, and floating-point mathematical operations (see '840, column 5, lines 47-53). Floating-point addition, subtraction, multiplication, division, and square root are supported in hardware ('840, column 15, lines 57-59). Similarly, the '599 patent describes structure of a general purpose, programmable processor for broadband applications, and the '599 patent includes and refers to a Microfiche Appendix (referred to herein as the Zeus manual) that describes, for example, an instruction set for the general purpose processor.

19. The Terpsichore manual describes all of the aforementioned elements of claims 9, 18, 33, 34, 40, and 41, on at least pages 19-21, 24-26, 29, 47-48, 129-131, and 150-157 (attached as Exhibit A). The Zeus manual describes all of the elements of claims 9, 18, 33, 34, 40, and 41, on at least pages 14-16, 19-21, 23-24, 55, 123-125, 128-130, and 258-260 (attached as Exhibit B).

Claims 9 and 18

20. The Terpsichore manual describes all of the aforementioned elements of claim 9 (and the substantially similar aforementioned elements of claim 18), on at least pages 19-21, 24-25, 29, 47-48, and 129-131, describing Group Floating-point instructions such as various forms of Group Floating-point Multiply instructions. Paragraphs 21-27 of this declaration discuss selected portions of those pages. Paragraphs 28-32 of this declaration discuss how the elements of claim 9 (and the substantially similar aforementioned elements of claim 18) are described by those pages. The Zeus manual describes all of the elements of claim 9 (and the substantially similar aforementioned elements of claim 18), on at least pages 14-16, 19-20, 23-24, 55, and 258-260.

21. The '840 patent describes various floating-point data sizes such as 16, 32, 64, and 128 bits (see '840, column 15, lines 62-65, and Fig. 9b, reproduced below). The Terpsichore manual, for example on pages 19-21, describes the various floating-point data sizes as designed to satisfy ANSI/IEEE standard 754-1985. Similarly, the Zeus manual, for example on pages 14-16, provides similar information.



22. The '840 patent provides description relating to floating-point hardware capabilities, such as in the Terpsichore manual on page 29, "operations supported in hardware are floating-point add, subtract, multiply, divide, and square root", and further on page 47, "partitioning favored for the initial implementation places all instructions that involving shifting and shuffling in one execution unit, and all instructions that involve multiplication, including fixed-point and floating-point multiply and add in another unit". Similarly, the Zeus manual, for example on pages 23-24 and 55, has similar descriptive information.

23. The Terpsichore manual describes several variations of Group Floating-point Multiply instructions, such as GF.MUL.16, GF.MUL.32, and GF.MUL.64, (among others) as described on pages 129-131, and reproduced below (excerpted and annotations added). Similarly, the Zeus manual, for example on pages 258-260, provides similar information.

Group Floating-point

These operations take two values from registers, perform floating-point arithmetic on groups of bits in the operands, and place the concatenated results in a register.

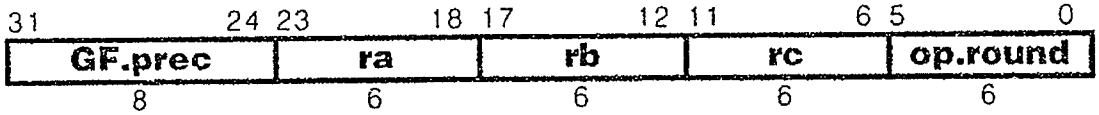
Operation codes

GF.MUL.16	Group floating-point multiply half
GF.MUL.16.C	Group floating-point multiply half ceiling
GF.MUL.16.F	Group floating-point multiply half floor
GF.MUL.16.N	Group floating-point multiply half nearest
GF.MUL.16.T	Group floating-point multiply half truncate
GF.MUL.16.X	Group floating-point multiply half exact
GF.MUL.32	Group floating-point multiply single
GF.MUL.32.C	Group floating-point multiply single ceiling
GF.MUL.32.F	Group floating-point multiply single floor
GF.MUL.32.N	Group floating-point multiply single nearest
GF.MUL.32.T	Group floating-point multiply single truncate
GF.MUL.32.X	Group floating-point multiply single exact
GF.MUL.64	Group floating-point multiply double

24. The Terpsichore manual, on page 130, describes an instruction format for the Group Floating-point instructions (including Multiply forms as well as Add and Divide forms), reproduced below. Similarly, the Zeus manual, for example on page 260, provides similar information.

Format

GF.op.prec.round rc=ra,rb

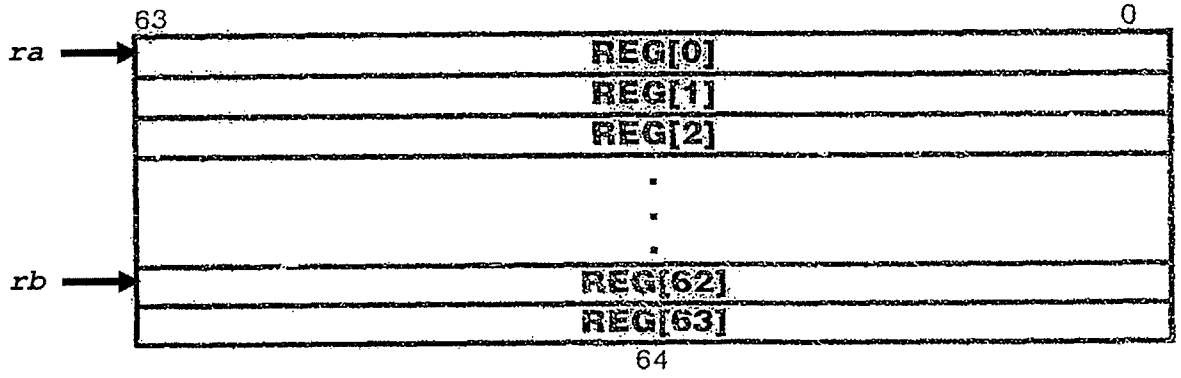


The operands of the Group Floating-point instructions (such as Multiply forms) include '*ra*', '*rb*', and '*rc*'. As described in more detail in paragraphs 26(b)-26(c)ii of this declaration, contents of registers specified by the '*ra*' and '*rb*' operands are interpreted as respective collections of partitioned floating-point operands. The partitioned floating-point operands relating to '*ra*' and '*rb*' are pairwise multiplied together, and the results are concatenated and then stored in a register specified by the '*rc*' operand.

25. The Terpsichore manual, on pages 24-25, describes registers referenced as operands of some instructions (such as Group Floating-point instructions, including Multiply forms as well as Add and Divide forms), as reproduced below (with annotations illustrating examples of an '*ra*' operand of '0' and an '*rb*' operand of '62'). Similarly the Zeus manual, for example on pages 19-20, provides similar information.

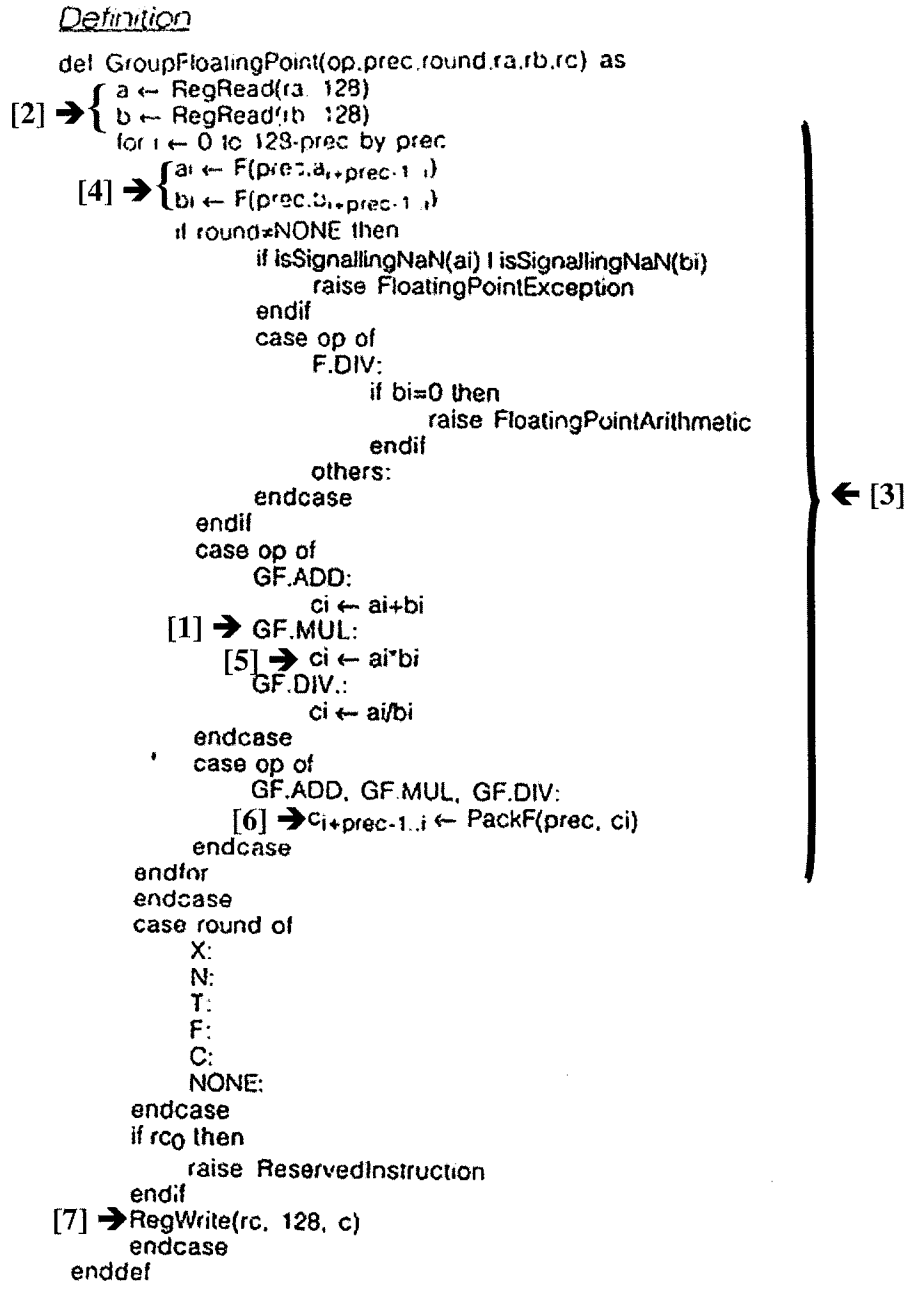
General Registers

Terpsichore user state includes 64 general registers. All are identical; there is no dedicated zero-valued register, and there are no dedicated floating-point registers.



The forgoing registers are included in register file 110 of Fig. 7 of the '840 patent.

26. The Terpsichore manual, on pages 130-131, describes a definition of various Group Floating-point instructions, including several Multiply forms. The definition is reproduced below, with annotations highlighting several elements that are discussed in the following sub-paragraphs concerning highlights of what one of ordinary skill in the art would understand from the description of the Terpsichore manual. Similarly, the Zeus manual, for example on page 260, provides similar information.



(a) The '*op*' field of the instruction is decoded to distinguish a Multiply form (MUL) from an Add (ADD) or Divide (DIV) form, for example, as highlighted by annotation [1].

(b) Source operands are read from pairs of registers, as specified by the '`ra`'and '`rb`' operands, into variables '*a*'and '*b*', respectively (see annotation [2]), such as including reading REG[0] into the least-significant 64 bits of '*a*' and REG[1] into the most-significant 64 bits of '*a*', when '`ra`' is 0.

(c) A '*for*' construct (see annotation [3]) specifies a number of evaluations of elements of the construct according to a '*prec*' operand that specifies a (floating-point) precision to interpret the operands.

i. For example, if the '*prec*' operand is 16, then the '*for*' construct is evaluated with eight values for variable '*i*' (0, 16, 32, 48, 64, 80, 96, and 112), respectively. For another example, if the '*prec*' operand is 64, then the construct is evaluated with two values for '*i*' (0 and 64), respectively.

ii. Each evaluation begins by determining a partitioned floating-point value from each of the variables '*a*' and '*b*' (see annotation [4]) in accordance with the '*prec*' operand. For example, if the '*prec*' operand is 16, then for the evaluation where '*i*' is 0, a first partitioned floating-point value is determined from '*a*' from the least-significant 16 bits of '*a*', or '$a_{15..0}$' as identified by the expression '$a_{i+prec-1..i}$' in the definition. Further in the evaluation where '*i*' is 0, a second floating-point value is determined from the least 16 bits of '*b*'. Continuing with the example, for the evaluation where '*i*' is 16, partitioned floating-point values are determined from the next-most-significant 16 bits of '*a*' and '*b*', such that bits 31 to 16 determine the floating-point values. Further continuing with the example, for the evaluation where '*i*' is 112, partitioned floating-point values are determined from the most significant 16 bits of '*a*' and '*b*'. For another example of construct evaluations, if the '*prec*' operand is 64, then the '*for*' construct is evaluated with two values for '*i*' (0 and 64). Partitioned floating-point values are determined in the evaluation where '*i*' is 0 as the least-significant 64 bits of '*a*' and '*b*', and in the evaluation where '*i*' is 64 as the most-significant 64 bits.

iii. The '*for*' construct processing continues by decoding a rounding mode and processing accordingly, and then decoding the '*op*' field of the instruction (see previously discussed annotation [1]).

iv. In the case of a Multiply form, the '*for*' construct processing continues by multiplying the partitioned floating-point values from '*a*' and '*b*' by each other (see annotation [5]). The multiplying is in accordance with floating-point multiplying.

v. The '*for*' construct processing completes by writing the result of the multiplies into appropriate bits of destination variable '*c*' (see annotation [6]) in accordance with the '*prec*' operand. The appropriate bit locations of '*c*' are identical to the bit locations of '*a*' and '*b*' that the partitioned floating-point values were determined from. For example, if the '*prec*' operand is 16, then for the evaluation where '*i*' is 0, the least-significant 16 bits (i.e. bits 15 to zero) of '*c*' are written, and for the evaluation where '*i*' is 16, the next-most-significant 16 bits (i.e. bits 31 to 16) of '*c*' are written. For the evaluation where '*i*' is 112, the most-significant bits (i.e. bits 127 to 112) of '*c*' are written.

vi. Note that each evaluation of the '*for*' construct is independent of the other evaluations, serving to operate on different unique and non-overlapping partitioned fields of the operands. Thus each evaluation is performable in parallel with the other evaluations, sequentially with the other evaluations, or any combination thereof.

(d) After completion of the '*for*' construct, processing completes by writing '*c*' into a pair of registers, as specified by the '*rc*' operand (see annotation [7]).

One of ordinary skill in the art would readily understand that the computation of the floating-point multiplies would occur in ALU 102 of Fig. 7.

27. Thus the Group Floating-point Multiply instructions are described in the Terpsichore manual (and also the Zeus manual) as interpreting contents of two source registers as respective pluralities of floating-point operands that are multiplied together, producing a plurality of products as results. The results are concatenated together and stored in a register specified by a third operand.

28. At least the Terpsichore manual pages 19-21, 24-25, 29, 47-48, and 129-131 describe all elements of claim 9 and substantially similar claim 18 (as amended), as described in more detail in paragraphs 29-32 of this declaration. Similarly, at least the Zeus manual pages 14-16, 19-20, 23-24, 55, and 258-260 describe all elements of claim 9 and substantially similar claim 18 (as amended).

29. The element (of claim 9) decoding a second single instruction specifying a register containing a first plurality of floating-point operands and another register containing a second plurality of floating-point operands is described by the Terpsichore manual, for example as annotated and discussed by paragraphs 21-27 of this declaration. Each of the Group Floating-point Multiply instructions is an exemplary single instruction that specifies registers containing respective pluralities of floating-point operands, via, for example, the '*ra*', and '*rb*' operands. See paragraphs 26(b) and 26(c)ii of this declaration for additional detailed discussion. As discussed in paragraph 22 of this declaration, an execution unit is clearly disclosed that is operable as claimed in this element of claim 9, as well as the other elements of claim 9 discussed in the following paragraphs.

30. The element multiplying the first plurality of floating-point operands by the second plurality of floating-point operands to produce a plurality of products is described by the Terpsichore manual, for example as annotated and discussed in paragraphs 21-27 of this declaration. Each of the Group Floating-point Multiply instructions is operable to perform a floating-point multiply on operands from registers specified by '*ra*' and '*rb*'. See paragraph 26(c)iv of this declaration for additional detailed discussion.

31. The element providing the plurality of products to partitioned fields of a result register as a catenated result is described by the Terpsichore manual, for example as annotated and discussed in paragraphs 21-27 of this declaration. Each of the Group Floating-point Multiply instructions is operable to produce products into fields of bits of result registers via, for

example, the '*rc*' operand. See paragraphs 26(c)v and 26(d) of this declaration for additional detailed discussion.

32. Thus every element of claim 9 and substantially similar claim 18 (as amended) are described at least by the Terpsichore manual on pages 19-21, 24-25, 29, 47-48, and 129-131. In addition, every element of claim 9 and substantially similar claim 18 are also described at least by the Zeus manual on pages 14-16 (describing floating-point data formats), pages 19-20 (describing general registers), pages 23-24 and 55 (describing floating-point arithmetic hardware), and pages 258-260 (a definition for Ensemble Floating-point instructions, including Multiply forms).

Claims 33 and 40

33. The Terpsichore manual describes all of the aforementioned elements of claim 33 (and the substantially similar aforementioned elements of claim 40), on at least pages 24-26 and 150-157, describing Store Immediate instructions such as various forms of Store Multiplex Immediate instructions. The Zeus manual describes all of the elements of claim 33 (and the substantially similar aforementioned elements of claim 40), on at least pages 19-21, 123-125, and 128-130.

34. Claim 33 is dependent upon claim 28, and context associated with the element (of claim 33) the first and second operands is from claim 28, reproduced below (as amended):

> {claim 28} A method for processing data in a programmable processor, the method comprising:
>
> decoding a single instruction for performing a bitwise insert operation on data in at least one register in a register file within the programmable processor, the bitwise insert operation operating on a first operand and a second operand stored in the at least one register in the register file, wherein each bit in the second operand is individually selectable as either having a first predetermined value or a second predetermined value; and
>
> for each bit in the first operand, the bitwise insert operation inserting the bit into a corresponding bit position in a destination value if a corresponding bit in the second operand has the first predetermined value.

35. As is described in more detail in paragraphs 36-39 of this declaration, the Terpsichore manual, on at least pages 24-26 and 150-157, describes all of the limitations of claim 28, in addition to claim 33, with respect to several variations of Store and Store Immediate instructions, including Store Multiplex and Store Multiplex Immediate forms. The initial Van Dyke declaration, in paragraphs 22-28, discusses selected portions of those pages. Similarly, at least the Zeus manual pages 19-21, 123-125, and 128-130 describe all elements of claims 28 and 33.

36. The element of (claim 28) decoding a single instruction for performing a bitwise insert operation on data in at least one register in a register file within the programmable processor, the bitwise insert operation operating on a first operand and a second operand stored

in the at least one register in the register file, wherein each bit in the second operand is individually selectable as either having a first predetermined value or a second predetermined value is described by the Terpsichore manual, as annotated and discussed in paragraphs 22-28 of the initial Van Dyke declaration. Each of the Store Multiplex Immediate instructions is a single instruction, as evidenced at least by the dedicated operation codes "S.MUX.64.B.A.I" and "S.MUX.64.L.A.I". Each of the Store Multiplex Immediate instructions is for performing a bitwise insert operation, since the combination of bit-wise logical-AND and bit-wise logical OR described for computing the store value results in insertion of a bit in place of another bit, e.g. "insertion" (see paragraph 27 of the initial Van Dyke declaration). The claimed at least one register corresponds to the register pair identified by the '*rb*' operand (see paragraph 26 of the initial Van Dyke declaration). The claimed first and second operands correspond respectively to the odd- and even-numbered registers of the register pair. There are no stated restrictions on values for the second operand, and therefore as claimed, each bit in the second operand is individually selectable as having either a first or a second predetermined value.

37. The element (of claim 28) for each bit in the first operand, the bitwise insert operation inserting the bit into a corresponding bit position in a destination value if a corresponding bit in the second operand has the first predetermined value is described by the Terpsichore manual, as annotated and discussed in paragraphs 22-28 of the initial Van Dyke declaration. As discussed in paragraph 27 of the initial Van Dyke declaration, a store value is determined by bit-wise multiplexing (e.g. "inserting") between a first data input and a second data input, based on a control input. The second data input is from memory. The first data input is the upper 64 bits of a value identified in the Terpsichore manual as '*m*', and the control input is the lower 64 bits of '*m*'. As discussed in paragraph 26 of the initial Van Dyke declaration, the upper 64 bits of '*m*' are obtained from the odd-numbered register identified by '*rb*', corresponding to the claimed first operand. Further the lower 64 bits of '*m*' are obtained from the even-numbered register identified by '*rb*', corresponding to the claimed second operand.

38. Therefore, according to the discussion in paragraphs 36-37 of this declaration, the element (of claim 33) wherein each of the first and second operands has a width of 64 bits the first operand corresponds to the upper 64 bits of '*m*' that are obtained from the odd-numbered register identified by '*rb*'. The second operand corresponds to the lower 64 bits of '*m*' that are obtained from the even-numbered register identified by '*rb*'. Thus both the first and the second operands are described as having a width of 64 bits.

39. Thus every element of claim 33 is described at least by the Terpsichore manual on pages 24-26 and 150-157. In addition, every element of claim 33 is also described at least by the Zeus manual on pages 19-21, 123-125, and 128-130.

40. Claim 40 is dependent upon claim 35, and context associated with the element (of claim 40) the first and second operands is from claim 35, reproduced below (as amended):

> {claim 35} A computer-readable storage medium having stored therein a plurality of instructions that cause a programmable processor to perform operations on data in the programmable processor, the plurality of instructions comprising:

an instruction that causes the processor to perform a bitwise insert operation on data in at least one register in a register file within the programmable processor, the bitwise insert operation operating on a first operand and a second operand stored in the at least one register in the register file, wherein each bit in the second operand is individually selectable as either having a first predetermined value or a second predetermined value; and

wherein for each bit in the first operand, the bitwise insert operation inserts the bit into a corresponding bit position in a destination value if a corresponding bit in the second operand has the first predetermined value.

41. Claim 35 (a computer-readable storage medium claim) is substantially similar to claim 28 (a method claim), and as previously discussed, claim 40 is substantially similar to claim 33. Therefore paragraphs 33-39 in this declaration concerning claim 33 and parent claim 28 are also applicable to claim 40 and parent claim 35. Thus every element of claim 40 is described at least by the Terpsichore manual on pages 24-26 and 150-157. In addition, every element of claim 40 is also described at least by the Zeus manual on pages 19-21, 123-125, and 128-130.

Claims 34 and 41

42. The Terpsichore manual describes all of the aforementioned elements of claim 34 (and the substantially similar aforementioned elements of claim 41), on at least pages 19-21, 24-25, 29, 47-48, and 129-131, describing Group Floating-point instructions such as various forms of Group Floating-point Add, Divide, and Multiply instructions. Paragraphs 43-50 of this declaration discuss selected portions of those pages. Paragraphs 51-52 of this declaration describe how the elements of claim 34 (and the substantially similar aforementioned elements of claim 41) are described by those pages. The Zeus manual describes all of the elements of claim 34 (and the substantially similar aforementioned elements of claim 41), on at least pages 14-16, 19-20, 23-24, 55, and 258-260.

43. As discussed in more detail in paragraph 21 of this declaration, the '840 and the '599 patents describe various floating-point data sizes.

44. As discussed in more detail in paragraph 22 of this declaration, the '840 and the '599 patents describe various floating-point hardware capabilities.

45. The Terpsichore manual describes several types and variations of Group Floating-point instructions, such as Group Floating-point Add (e.g.e GF.ADD.64), Divide (e.g. GF.DIV.32), and Multiply (e.g. GF.MUL.16) instructions, as described on pages 129-131, and reproduced below (excerpted and annotations added). Similarly, the Zeus manual, for example on pages 258-260, provides similar information.

Group Floating-point

These operations take two values from registers, perform floating-point arithmetic on groups of bits in the operands, and place the concatenated results in a register.

Operation codes

GF.ADD.64	Group floating-point add double
GF.ADD.64 .C	Group floating-point add double ceiling
GF.ADD.64 .F	Group floating-point add double floor
GF.ADD.64 .N	Group floating-point add double nearest
GF.ADD.64 .T	Group floating-point add double truncate
GF.ADD.64 .X	Group floating-point add double exact
GF.DIV.16	Group floating-point divide half
GF.DIV.16.C	Group floating-point divide half ceiling
GF.DIV.16.F	Group floating-point divide half floor
GF.DIV.16.N	Group floating-point divide half nearest
GF.DIV.16.T	Group floating-point divide half truncate
GF.DIV.16.X	Group floating-point divide half exact
GF.DIV.32	Group floating-point divide single
GF.DIV.32.C	Group floating-point divide single ceiling
GF.DIV.32.F	Group floating-point divide single floor
GF.DIV.32.N	Group floating-point divide single nearest
GF.DIV.32.T	Group floating-point divide single truncate
GF.DIV.32.X	Group floating-point divide single exact
GF.DIV.64	Group floating-point divide double
GF.DIV.64.C	Group floating-point divide double ceiling
GF.DIV.64.F	Group floating-point divide double floor
GF.DIV.64.N	Group floating-point divide double nearest
GF.DIV.64.T	Group floating-point divide double truncate
GF.DIV.64.X	Group floating-point divide double exact
GF.MUL.16	Group floating-point multiply half
GF.MUL.16.C	Group floating-point multiply half ceiling
GF.MUL.16.F	Group floating-point multiply half floor

46. As discussed in paragraph 24 of this declaration, the '840 and the '599 patents describe an instruction format for the Group Floating-point instructions (including Add, Divide, and Multiply forms).

47. As discussed in paragraph 25 of this declaration, the '840 and the '599 patents describe registers referenced as operands of some instructions (such as Group Floating-point Add, Divide, and Multiply instructions).

48. As discussed in paragraph 26 of this declaration, the '840 patent and the '599 patent describe definitions of various Group Floating-point instructions, including several Multiply forms. In addition, Add and Divide forms of Group Floating-point instructions are described by the '840 patent and the '599 patent, as evidenced by the following excerpt from page 131 of the Terpsichore manual:

```
case op of
    GF.ADD:
        ci ← ai+bi
    GF.MUL:
        ci ← ai*bi
    GF.DIV.:
        ci ← ai/bi
endcase
```

Similarly, the Zeus manual, for example on page 260, provides similar information.

49. The discussion of Multiply forms of Group Floating-point instructions in sub-paragraphs 26(a)-26(d) of this declaration is generally applicable to Add and Divide forms, as well. Rather than multiplying the partitioned floating-point values (as discussed in sub-paragraph 26(c)iv of this declaration), the floating-point values are added (Add form) or divided (Divide form).

50. Thus the Group Floating-point instructions of Add, Divide, and Multiply forms are described in the Terpsichore manual (and also the Zeus manual) as interpreting contents of two source registers as respective pluralities of floating-point operands that are added, divided, or multiplied together, respectively, producing a plurality of floating-point results that are concatenated together and stored in a register specified by a third operand.

51. The element (of claim 34) a step of executing a plurality of different group floating-point arithmetic operations that arithmetically operate on multiple floating-point operands stored in partitioned fields of registers in the register file to produce a catenated result that is returned to a register in the register file, wherein the catetated result comprises a plurality of individual floating-point results is described by the Terpsichore manual, for example as annotated and discussed in paragraphs 43-50 of this declaration. The Add, Divide, and Multiply forms of Group Floating-point instructions are exemplary instructions that embody arithmetic operations that operate on multiple floating-point operands, for example as specified by the '*ra*', and '*rb*' operands. Results of the arithmetic operations are returned, for example, to result registers specified by the '*rc*' operand. As discussed in paragraph 22 of this declaration, an execution unit is clearly disclosed that is operable as claimed in claim 25.

52. Thus every element of claim 34 and substantially similar claim 41 (as amended) are described at least by the Terpsichore manual on pages 19-21, 24-25, 29, 47-48, and 129-131. In addition, every element of claim 34 and substantially similar claim 41 are also described at

least by the Zeus manual on pages 14-16 (describing floating-point data formats), pages 19-20 (describing general registers), pages 23-24 and 55 (describing floating-point arithmetic hardware), and pages 258-260 (a definition for Ensemble Floating-point instructions, including Add, Divide, and Multiply forms).

Summary and Closing:

53. The '840 patent, including the Terpsichore manual, provides sufficient information in sufficient detail describing the claimed invention (as amended) of the 10/757,866 patent application, that one of ordinary skill in the art would reasonably conclude that the inventors had possession of the claimed invention at the time of filing the '840 patent. Further, the '840 patent, including the Terpsichore manual, provides sufficient information regarding the subject matter of the claimed invention (as amended) of the 10/757,866 patent application to enable one of ordinary skill in the pertinent art to make and use the claimed invention without undue experimentation. In addition, the '599 patent, including the Zeus manual, provides sufficient information in sufficient detail describing the claimed invention (as amended) of the 10/757,866 patent application, that one of ordinary skill in the art would reasonably conclude that the inventors had possession of the claimed invention at the time of filing the '599 patent. Further, the '599 patent, including the Zeus manual, provides sufficient information regarding the subject matter of the claimed invention (as amended) of the 10/757,866 patent application to enable one of ordinary skill in the pertinent art to make and use the claimed invention without undue experimentation.

54. Therefore, I believe that each of the '840 patent, including the Terpsichore manual, and the '599 patent, including the Zeus manual, provide adequate written description and enablement as required by 35 USC § 112 for the limitations of claims 9, 18, 33, 34, 40, and 41 (as amended) of the 10/757,866 patent application, as discussed in paragraph 3 of this declaration.

55. I have had no communication with any of the inventors of the 10/757,866 patent application (Craig Hansen and John Moussouris) relating to any material in this declaration.

56. I have been hired as a consultant in connection with procedures before the United States Patent and Trademark Office (USPTO) regarding patents and patent applications assigned to Microunity Systems Engineering, Inc., including the media processor patent application. I am being compensated for my services at the rate of $325/hour. Other than acting as a consultant in connection with procedures before the USPTO, I have no interest or connection with Microunity Systems Engineering, Inc.

57. During my evaluation of the media processor patent application, I have been impressed by the thoroughness and overall high-quality of the Terpsichore and Zeus manuals. The manuals provide clear and unambiguous descriptions of media processing systems and are thorough and well-written. The manuals provide comprehensive descriptions of instructions in complete architectural detail. The information in the manuals would have been readily understood and easily accessible to software engineers coding the media processing systems, and hardware engineers implementing microprocessors for use in the media processing systems, and that is exactly what architecture reference manuals should be. This is not surprising, since the

'840 patent and the '599 patent each include an architecture manual that is intended to enable hardware engineers to do exactly that – design, build, and implement a media processor that would include circuitry for the claim limitations set forth in paragraph 3 of this declaration, as described in the Terpsichore and the Zeus architecture manuals.

58. I hereby declare that all statements made herein are of my own knowledge are true and that all statements made on information and belief are believed to be true; and further that these statements were made with the knowledge that willful false statements and the like so made are punishable by fine or imprisonment, or both, under Section 1001 Title 18 of the United States Code and that such willful false statements may jeopardize the validity of the application or any patent issuing therefrom.

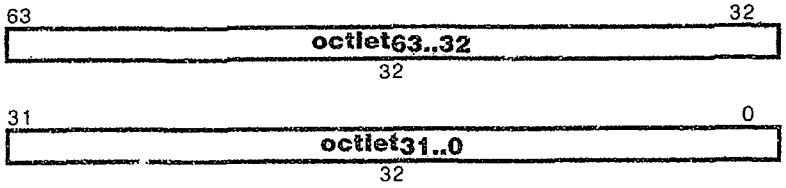
Date: 5/2/2008

Korbin Van Dyke: K. Van Dyke

Address: 3343 Little Valley Rd.
Sunol, CA 94586

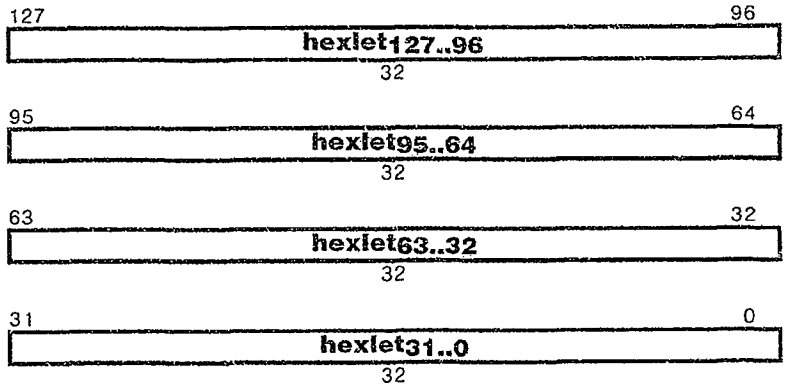
Octlet

An octlet is the catenation of 64 bits, and is the concatenation of eight bytes:



Hexlet

A hexlet is the catenation of 128 bits, and is the concatenation of sixteen bytes:



Address

Terpsichore addresses are octlet quantities.

Floating-point Data

Terpsichore's floating-point formats are designed to satisfy ANSI/IEEE standard 754-1985: Binary Floating-point Arithmetic. Standard 754 leaves certain aspects to the discretion of the implementor:

Terpsichore adds additional half-precision and quad-precision formats to standard 754's single-precision and double-precision formats. Terpsichore's double-precision satisfies standard 754's precision requirements for a single-extended format, and Terpsichore's quad-precision satisfies standard 754's precision requirements for a double-extended format.

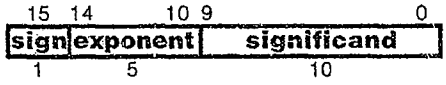
Quiet NaN values are denoted by any sign bit value, an exponent field of all one bits, and a non-zero significand with the most significant bit cleared. Quiet NaN

values generated by default exception handling of standard operations have a zero sign bit, an exponent field of all one bits, and a significand field with the most significant bit cleared, the next-most significant bit set, and all other bits cleared.

Signaling NaN values are denoted by any sign bit value, an exponent field of all one bits, and a non-zero significand with the most significant bit set.

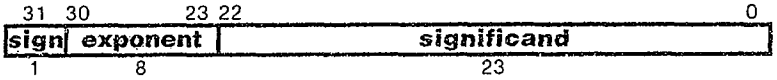
Half-precision Floating-point

Terpsichore half precision uses a format similar to standard 754's requirements, reduced to a 16-bit overall format. The format contains sufficient precision and exponent range to hold a 12-bit signed integer.



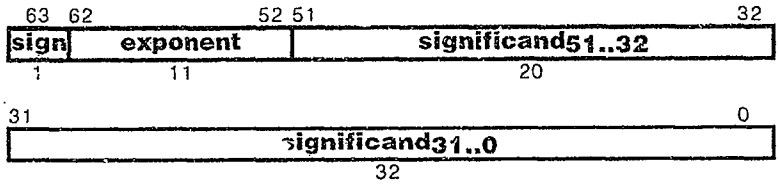
Single-precision Floating-point

Terpsichore single precision satisfies standard 754's requirements for "single."



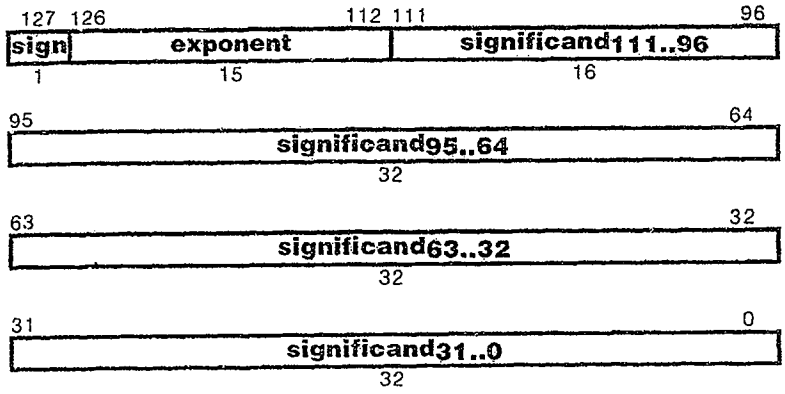
Double-precision Floating-point

Terpsichore double precision satisfies standard 754's requirements for "double."

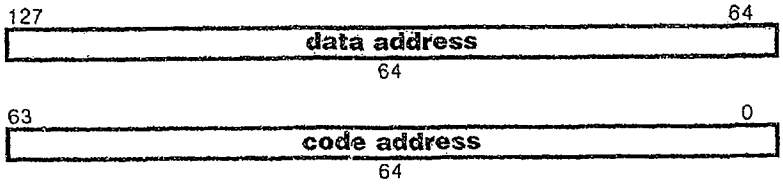


Quad-precision Floating-point

Terpsichore quad precision satisfies standard 754's requirements for "double extended," but has additional significand precision to use 128 bits.



The gateway contains two data items within its structure, a code address and a data address:

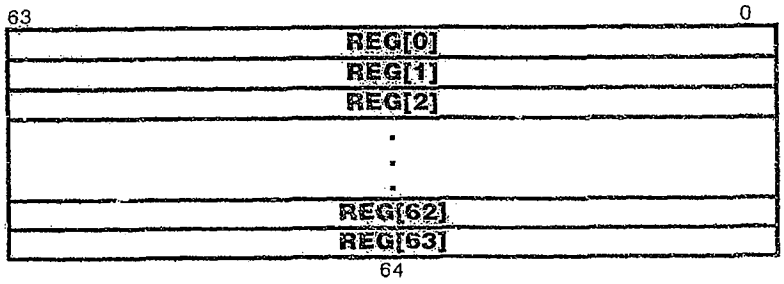


User state

The user state consists of hardware data structures that are accessible to all conventional compiled code. The Terpsichore user state is designed to be as regular as possible, and consists only of the general registers, the program counter, and virtual memory. There are no specialized registers for condition codes, operating modes, rounding modes, integer multiply/divide, or floating-point values.

General Registers

Terpsichore user state includes 64 general registers. All are identical; there is no dedicated zero-valued register, and there are no dedicated floating-point registers.



Definition

```
def val ← RegRead(rn. size)
    case size of
        64:
            val ← REG[rn]
        128:
            if rn0 then
                raise ReservedInstruction
            endif
            val ← REG[rn+1] || REG[rn]
    endcase
enddef
```

```
def RegWrite(rn, size, val)
    case size of
        64:
            REG[rn] ← val63..0
        128:
            if rn0 then
                raise ReservedInstruction
            endif
            REG[rn+1] ← val127..64
            REG[rn] ← val63..0
    endcase
enddef
```

Program Counter

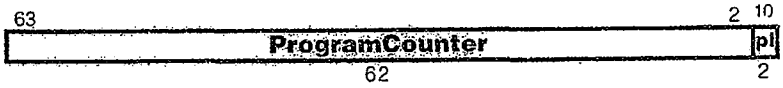
The program counter contains the address of the currently executing instruction. This register is implicitly manipulated by branch instructions, and read by branch instructions that save a return address in a general register.

Privilege Level

The privilege level register contains the privilege level of the currently executing instruction. This register is implicitly manipulated by branch gateway and branch down instructions, and read by branch gateway instructions that save a return address in a general register.

Program Counter and Privilege Level

The program counter and privilege level may be packed into a single octlet. This combined data structure is saved by the Branch Gateway instruction and restored by the Branch Down instruction.



System state

The system state consists of the facilities not normally used by conventional compiled code. These facilities provide mechanisms to execute such code in a fully virtual environment. All system state is memory mapped, so that it can be manipulated by compiled code.

Fixed-point

Terpsichore provides load and store instructions to move data between memory and the registers, branch instructions to compare the contents of registers and to transfer control from one code address to another, and arithmetic operations to perform computation on the contents of registers, returning the result to registers.

Load and Store

The load and store instructions move data between memory and the registers. When loading data from memory into a register, values are zero-extended or sign-extended to fill the register. When storing data from a register into memory, values are truncated on the left to fit the specified memory region.

Load and store instructions that specify a memory region of more than one byte may use either little-endian or big-endian byte ordering: the size and ordering are explicitly specified in the instruction. Regions larger than one byte may be either aligned to addresses that are an even multiple of the size of the region, or of unspecified alignment: alignment checking is also explicitly specified in the instruction.

The load and store instructions are used for fixed-point data as well as floating-point and digital signal processing data; Terpsichore has a single bank of registers for all data types.

Swap instructions provide multithread and multiprocessor synchronization, using indivisable operations: add-and-swap, compare-and-swap, and multiplex-and-swap. A store-multiplex operation provides the ability to indivisably write to a portion of an octlet. These instructions always operate on aligned octlet data, using either little-endian or big-endian byte ordering.

Branch Conditionally

The fixed-point compare-and-branch instructions provide all arithmetic tests for equality and inequality of signed and unsigned fixed-point values. Tests are performed either between two operands contained in general registers, or on the bitwise and of two operands. Depending on the result of the compare, either a branch is taken, or not taken. A taken branch causes an immediate transfer of the program counter to the target of the branch, specified by a 12-bit signed offset from the location of the branch instruction. A non-taken branch causes no transfer; execution continues with the following instruction.

Branch Unconditionally

Other branch instructions provide for unconditional transfer of control to addresses too distant to be reached by a 12-bit offset, and to transfer to a target while placing the location following the branch into a register. The branch through gateway instruction provides a secure means to access code at a higher privilege level, in a form similar to a normal procedure call.

Arithmetic Operations

The operations supported in hardware are floating-point add, subtract, multiply, divide, and square root. Other operations required by the ANSI/IEEE floating-point standard are provided by software libraries.

The operations explicitly specify the precision of the operation, and round the result to the specified precision at the conclusion of each operation.

A single instruction provides a floating-point multiply with the result fed into a floating-point add. The result is computed as if the multiply is performed to infinite precision, added as if in infinite precision, then rounded. This operation is a particularly good match to the needs of vector linear algebra routines.

Rounding

Rounding is specified within the instructions explicitly, to avoid maintaining explicit state for a rounding mode.

Exceptions

All the mandated floating-point exception conditions cause a trap when they occur; maintenance of sticky and other status bits may be performed using software routines. Because the floating-point inexact exception may be very frequent, this exception only occurs when specified in the instruction explicitly. Arithmetic operations may also specify that all exceptions are to be handled by default, generating special results instead of traps.

Digital Signal Processing

The Terpsichore processor provides a set of operations that maintain the fullest possible use of 64- and 128-bit data paths when operating on lower-precision fixed-point or floating-point vector values. These operations are useful for several application areas, including digital signal processing, image processing, and synthetic graphics. The basic goal of these operations is to accelerate the performance of algorithms that exhibit the following characteristics:

Low-precision arithmetic

The operands and intermediate results are fixed-point values represented in no greater than 64 bit precision. For floating-point arithmetic, operands and intermediate results are of 16, 32, or 64 bit precision.

The use of fixed-point arithmetic permits various forms of operation reordering that are not permitted in floating-point arithmetic. Specifically, commutativity and associativity, and distribution identities can be used to reorder operations. Compilers can evaluate operations to determine what intermediate precision is required to get the specified arithmetic result.

branch predicts that a future execution of the same branch will be taken. More elaborate prediction may cache the source and target addresses of multiple branches, both conditional and unconditional, and both forward and reverse.

The hardware prediction mechanism is tuned for optimizing conditional branches that close loops or express frequent alternatives, and will generally require substantially more cycles when executing conditional branches whose outcome is not predominately taken or not-taken. For such cases of unpredictable conditional results, the use of code which avoids conditional branches in favor of the use of set on compare and multiplex instructions may result in greater performance.

Where the above technique may not be applicable, a Euterpe pipeline may ensure that conditional branches which have a small positive offset be handled as if the branch is always predicted to be not taken, with the recovery of a misprediction causing cancellation of the instructions which have already been issued but not completed which would be skipped over by the taken conditional branch. This "conditional-skip" optimization is performed by the Euterpe implementation and requires no specific architectural feature to access or implement.

A Euterpe pipeline may also perform "branch-return" optimization, in which a branch-and-link instruction saves a branch target address which is used to predict the target of the next branch-register instruction. This optimization may be implemented with a depth of one (only one return address kept), or as a stack of finite depth, where a branch and link pushes onto the stack, and a branch-register pops from the stack. This optimization can eliminate the misprediction cost of simple procedure calls, as the calling branch is susceptible to hardware prediction, and the returning branch is predictable by the branch-return optimization. Like the conditional-skip optimization described above, this feature is performed by the Euterpe implementation and requires no specific architectural feature to access or implement.

Additional Load and Execute Resources

Studies of the dynamic distribution of Euterpe instructions on various benchmark suites indicate that the most frequently-issued instruction classes are load instructions and execute instructions. In a high-performance Euterpe implementation, it is advantageous to consider execution pipelines in which the ability to target the machine resources toward issuing load and execute instructions is increased.

One of the means to increase the ability to issue execute-class instructions is to provide the means to issue two execute instructions in a single-issue string. The execution unit actually requires several distinct resources, so by partitioning these resources, the issue capability can be increased without increasing the number of functional units, other than the increased register file read and write ports. The partitioning favored for the initial implementation places all instructions that involve shifting and shuffling in one execution unit, and all instructions that involve multiplication, including fixed-point and floating-point multiply and add in another unit. Resources used for implementing add, subtract, and bitwise logical operations may be duplicated, being modest in size compared to the shift and

multiply units, or shared between the two units, as the operations have low-enough latency that two operations might be pipelined within a single issue cycle. These instructions must generally be independent, except perhaps that two simple add, subtract, or bitwise logical may be performed dependently, if the resources for executing simple instructions are shared between the execution units.

One of the means to increase the ability to issue load-class instructions is to provide the means to issue two load instructions in a single-issue string. This would generally increase the resources required of the data fetch unit and the data cache, but a compensating solution is to steal the resources for the store instruction to execute the second load instruction. Thus, a single-issue string can then contain either two load instructions, or one load instruction and one store instruction, which uses the same register read ports and address computation resources as the basic 5-instruction string. This capability also may be employed to provide support for unaligned load and store instructions, where a single-issue string may contain as an alternative a single unaligned load or store instruction which uses the resources of the two load-class units in concert to accomplish the unaligned memory operation.

Result Forwarding

When temporally adjacent instructions are executed by seperate resources, the results of the first instruction must generally be forwarded directly to the resource used to execute the second instruction, where the result replaces a value which may have been fetched from a register file. Such forwarding paths use significant resources. A Terpsichore implementation must generally provide forwarding resources so that dependencies from earlier instructions within a string are immediately forwarded to later instructions, except between a first and second execution instruction as described above. In addition, when forwarding results from the execution units back to the data fetch unit, additional delay may be incurred.

Group Floating-point

These operations take two values from registers, perform floating-point arithmetic on groups of bits in the operands, and place the concatenated results in a register.

Operation codes

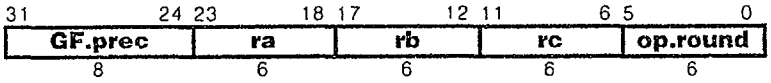
GF.ADD.16	Group floating-point add half
GF.ADD.16.C	Group floating-point add half ceiling
GF.ADD.16.F	Group floating-point add half floor
GF.ADD.16.N	Group floating-point add half nearest
GF.ADD.16.T	Group floating-point add half truncate
GF.ADD.16.X	Group floating-point add half exact
GF.ADD.32	Group floating-point add single
GF.ADD.32.C	Group floating-point add single ceiling
GF.ADD.32.F	Group floating-point add single floor
GF.ADD.32.N	Group floating-point add single nearest
GF.ADD.32.T	Group floating-point add single truncate
GF.ADD.32.X	Group floating-point add single exact
GF.ADD.64	Group floating-point add double
GF.ADD.64 .C	Group floating-point add double ceiling
GF.ADD.64 .F	Group floating-point add double floor
GF.ADD.64 .N	Group floating-point add double nearest
GF.ADD.64 .T	Group floating-point add double truncate
GF.ADD.64 .X	Group floating-point add double exact
GF.DIV.16	Group floating-point divide half
GF.DIV.16.C	Group floating-point divide half ceiling
GF.DIV.16.F	Group floating-point divide half floor
GF.DIV.16.N	Group floating-point divide half nearest
GF.DIV.16.T	Group floating-point divide half truncate
GF.DIV.16.X	Group floating-point divide half exact
GF.DIV.32	Group floating-point divide single
GF.DIV.32.C	Group floating-point divide single ceiling
GF.DIV.32.F	Group floating-point divide single floor
GF.DIV.32.N	Group floating-point divide single nearest
GF.DIV.32.T	Group floating-point divide single truncate
GF.DIV.32.X	Group floating-point divide single exact
GF.DIV.64	Group floating-point divide double
GF.DIV.64.C	Group floating-point divide double ceiling
GF.DIV.64.F	Group floating-point divide double floor
GF.DIV.64.N	Group floating-point divide double nearest
GF.DIV.64.T	Group floating-point divide double truncate
GF.DIV.64.X	Group floating-point divide double exact
GF.MUL.16	Group floating-point multiply half
GF.MUL.16.C	Group floating-point multiply half ceiling
GF.MUL.16.F	Group floating-point multiply half floor

GF.MUL.16.N	Group floating-point multiply half nearest
GF.MUL.16.T	Group floating-point multiply half truncate
GF.MUL.16.X	Group floating-point multiply half exact
GF.MUL.32	Group floating-point multiply single
GF.MUL.32.C	Group floating-point multiply single ceiling
GF.MUL.32.F	Group floating-point multiply single floor
GF.MUL.32.N	Group floating-point multiply single nearest
GF.MUL.32.T	Group floating-point multiply single truncate
GF.MUL.32.X	Group floating-point multiply single exact
GF.MUL.64	Group floating-point multiply double
GF.MUL.64.C	Group floating-point multiply double ceiling
GF.MUL.64.F	Group floating-point multiply double floor
GF.MUL.64.N	Group floating-point multiply double nearest
GF.MUL.64.T	Group floating-point multiply double truncate
GF.MUL.64.X	Group floating-point multiply double exact

	op	prec	round/trap
add	ADD	16 32 64 128	NONE C F N T X
divide	DIV	16 32 64 128	NONE C F N T X
multiply	MUL	16 32 64 128	NONE C F N T X

Format

GF.op.prec.round rc=ra,rb



Description

The contents of registers ra and rb are combined using the specified floating-point operation. The result is placed in register rc. The operation is rounded using the specified rounding option or using round-to-nearest if not specified. If a rounding option is specified, the operation raises a floating-point exception if a floating-point invalid operation, divide by zero, overflow, or underflow occurs, or when specified, if the result is inexact. If a rounding option is not specified, floating-point exceptions are not raised, and are handled according to the default rules of IEEE 754.

Definition

```
def GroupFloatingPoint(op.prec.round.ra.rb.rc) as
    a ← RegRead(ra, 128)
    b ← RegRead(rb, 128)
    for i ← 0 to 128-prec by prec
        ai ← F(prec.a[i+prec-1:i])
        bi ← F(prec.b[i+prec-1:i])
        if round≠NONE then
```

```
            if isSignallingNaN(ai) | isSignallingNaN(bi)
                raise FloatingPointException
            endif
            case op of
                F.DIV:
                    if bi=0 then
                        raise FloatingPointArithmetic
                    endif
                others:
            endcase
        endif
        case op of
            GF.ADD:
                ci ← ai+bi
            GF.MUL:
                ci ← ai*bi
            GF.DIV.:
                ci ← ai/bi
        endcase
        case op of
            GF.ADD, GF.MUL, GF.DIV:
                ci+prec-1..i ← PackF(prec, ci)
        endcase
    endfor
    endcase
    case round of
        X:
        N:
        T:
        F:
        C:
        NONE:
    endcase
    if rc0 then
        raise ReservedInstruction
    endif
    RegWrite(rc, 128, c)
    endcase
enddef
```

Exceptions

Reserved instruction
Floating-point arithmetic

Store

These operations add the contents of two registers to produce a virtual address, and store the contents of a register into memory.

Operation codes

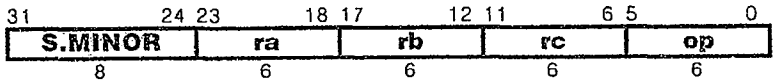
S.8[46]	Store byte
S.16.B	Store double big-endian
S.16.B.A	Store double big-endian aligned
S.16.L	Store double little-endian
S.16.L.A	Store double little-endian aligned
S.32.B	Store quadlet big-endian
S.32.B.A	Store quadlet big-endian aligned
S.32.L	Store quadlet little-endian
S.32.L.A	Store quadlet little-endian aligned
S.64.B	Store octlet big-endian
S.64.B.A	Store octlet big-endian aligned
S.64.L	Store octlet little-endian
S.64.L.A	Store octlet little-endian aligned
S.128.B	Store hexlet big-endian
S.128.B.A	Store hexlet big-endian aligned
S.128.L	Store hexlet little-endian
S.128.L.A	Store hexlet little-endian aligned
S.AAS.64.B.A	Store add-and-swap octlet big-endian aligned
S.AAS.64.L.A	Store add-and-swap octlet little-endian aligned
S.CAS.64.B.A	Store compare-and-swap octlet big-endian aligned
S.CAS.64.L.A	Store compare-and-swap octlet little-endian aligned
S.MAS.64.B.A	Store multiplex-and-swap octlet big-endian aligned
S.MAS.64.L.A	Store multiplex-and-swap octlet little-endian aligned
S.MUX.64.B.A	Store multiplex octlet big-endian aligned
S.MUX.64.L.A	Store multiplex octlet little-endian aligned

size				ordering		alignment
8						
16	32	64	128	L	B	
16	32	64	128	L	B	A

[46] S.8 need not specify byte ordering, nor need it specify alignment checking, as it stores a single byte.

Format

op ra,rb,rc



Description

A virtual address is computed from the sum of the contents of register ra and register rb. The contents of register rc, treated as the size specified, is stored in memory using the specified byte order.

If alignment is specified, the computed virtual address must be aligned, that is, it must be an exact multiple of the size expressed in bytes. If the address is not aligned an "access disallowed by virtual address" exception occurs.

Definition

```
def Store(op,ra,rb,rc) as
    case op of
        S8,
        S16L, S16LA, S16B, S16BA,
        S32L, S32LA, S32B, S32BA,
        S64L, S64LA, S64B, S64BA,
        S128L, S128LA, S128B, S128BA:
            function ← NONE
        SAAS64BA, SAAS64LA:
            function ← AAS
        SCAS64BA, SCAS64LA:
            function ← CAS
        SMAS64BA, SMAS64LA:
            function ← MAS
        SMUX64BA, SMUX64LA:
            function ← MUX
    endcase
    case op of
        S8:
            size ← 8
        S16L, S16LA, S16B, S16BA:
            size ← 16
        S32L, S32LA, S32B, S32BA:
            size ← 32
        S64L, S64LA, S64B, S64BA,
        SAAS64BA, SAAS64LA:
            size ← 64
        SCAS64BA, SCAS64LA, SMAS64BA, SMAS64LA, SMUX64BA, SMUX64LA:
            size ← 64
        S128L, S128LA, S128B, S128BA:
            size ← 128
    endcase
    case op of
        S8,
        S16L, S16LA, S16B, S16BA,
        S32L, S32LA, S32B, S32BA,
```


D 114

```
        S64L, S64LA, S64B, S64BA,
        SAAS64BA, SAAS64LA:
            rsize ← 64
        SCAS64BA, SCAS64LA  SMAS64BA, SMAS64LA, SMUX64BA, SMUX64LA:
            rsize ← 128
        S128L, S128LA, S128B, S128BA:
            rsize ← 128
    endcase
    case op of
        S8:
            align ← undefined
        S16L, S32L, S64L, S128L,
        S16B, S32B, S64B, S128B:
            align ← false
        S16LA, S32LA, S64LA, S128LA,
        S16BA, S32BA, S64BA, S128BA,
        SAAS64BA, SAAS64LA, SCAS64BA, SCAS64LA,
        SMAS64BA, SMAS64LA, SMUX64BA, SMUX64LA:
            align ← true
    endcase
    case op of
        S8:
            order ← undefined
        S16L, S32L, S64L, S128L,
        S16LA, S32LA, S64LA, S128LA,
        SAAS64LA, SCAS64LA, SMAS64LA, SMUX64LAI:
            order ← L
        S16B, S32B, S64B, S128B,
        S16BA, S32BA, S64BA, S128BA,
        SAAS64BA, SCAS64BA, SMAS64BA, SMUX64BAI:
            order ← B
    endcase
    a ← RegRead(ra, 64)
    b ← RegRead(rb, 64)
    VirtAddr ← a + b
    if align then
        if (VirtAddr and ((size/8)-1)) ≠ 0 then
            raise AccessDisallowedByVirtualAddress
        endif
    endif
    m ← RegRead(rc, rsize)
    case function of
        NONE:
            StoreMemory(VirtAddr,size,order,m_size-1..0)
        AAS:
            c ← LoadMemory(VirtAddr,size,order)
            StoreMemory(VirtAddr,size,order,m_63..0+c)
            RegWrite(rc, 64, c)
        CAS:
            c ← LoadMemory(VirtAddr,size,order)
            if (c = m_63..0) then
                StoreMemory(VirtAddr,size,order,m_127..64)
            endif
            RegWrite(rc, 64, c)
        MAS:
            c ← LoadMemory(VirtAddr,size,order)
            n ← (m_127..64 & m_63..0) | (c & ~m_63..0)
            StoreMemory(VirtAddr,size,order,n)
```


EX 1

```
            RegWrite(rc, 64, c)
        MUX:
            c ← LoadMemory(VirtAddr,size,order)
            n ← (m127..64 & m63..0) | (c & ~m63..0)
            StoreMemory(VirtAddr,size,order,n)
    endcase
enddef
```

Exceptions

Reserved instruction
Access disallowed by virtual address
Access disallowed by tag
Access disallowed by global TLB
Access disallowed by local TLB
Access detail required by tag
Access detail required by local TLB
Access detail required by global TLB
Cache coherence intervention required by tag
Cache coherence intervention required by local TLB
Cache coherence intervention required by global TLB
Local TLB miss
Global TLB miss

Store Immediate

These operations add the contents of a register to a sign-extended immediate value to produce a virtual address, and store the contents of a registerinto memory.

Operation codes

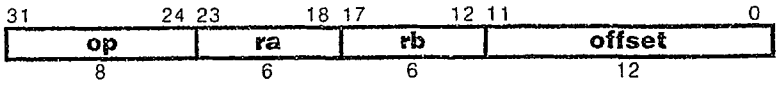
S.8.I[47]	Store byte immediate
S.16.B.A.I	Store double big-endian aligned immediate
S.16.B.I	Store double big-endian immediate
S.16.L.A.I	Store double little-endian aligned immediate
S.16.L.I	Store double little-endian immediate
S.32.B.A.I	Store quadlet big-endian aligned immediate
S.32.B.I	Store quadlet big-endian immediate
S.32.L.A.I	Store quadlet little-endian aligned immediate
S.32.L.I	Store quadlet little-endian immediate
S.64.B.A.I	Store octlet big-endian aligned immediate
S.64.B.I	Store octlet big-endian immediate
S.64.L.A.I	Store octlet little-endian aligned immediate
S.64.L.I	Store octlet little-endian immediate
S.128.B.A.I	Store hexlet big-endian aligned immediate
S.128.B.I	Store hexlet big-endian immediate
S.128.L.A.I	Store hexlet little-endian aligned immediate
S.128.L.I	Store hexlet little-endian immediate
S.AAS.64.B.A.I	Store add-and-swap octlet big-endian aligned immediate
S.AAS.64.L.A.I	Store add-and-swap octlet little-endian aligned immediate
S.CAS.64.B.A.I	Store compare-and-swap octlet big-endian aligned immediate
S.CAS.64.L.A.I	Store compare-and-swap octlet little-endian aligned immediate
S.MAS.64.B.A.I	Store multiplex-and-swap octlet big-endian aligned immediate
S.MAS.64.L.A.I	Store multiplex-and-swap octlet little-endian aligned immediate
S.MUX.64.B.A.I	Store multiplex octlet big-endian aligned immediate
S.MUX.64.L.A.I	Store multiplex octlet little-endian aligned immediate

size				ordering		alignment
8						
16	32	64	128	L	B	
16	32	64	128	L	B	A

[47] S.8.I need not specify byte ordering, nor need it specify alignment checking, as it stores a single byte.

Format

S.size.order.align.I ra,rb,offset



Description

A virtual address is computed from the sum of the contents of register ra and the sign-extended vale of the offset field. The contents of register rb, treated as the size specified, is stored in memory using the specified byte order.

If alignment is specified, the computed virtual address must be aligned, that is, it must be an exact multiple of the size expressed in bytes. If the address is not aligned an "access disallowed by virtual address" exception occurs.

Definition

```
def StoreImmediate(op,ra,rb,offset) as
    case op of
        S8I,
        S16LI, S16LAI, S16BI, S16BAI,
        S32LI, S32LAI, S32BI, S32BAI,
        S64LI, S64LAI, S64BI, S64BAI,
        S128LI, S128LAI, S128BI, S128BAI:
            function ← NONE
        SAAS64BAI, SAAS64LAI:
            function ← AAS
        SCAS64BAI, SCAS64LAI:
            function ← CAS
        SMAS64BAI, SMAS64LAI:
            function ← MAS
        SMUX64BAI, SMUX64LAI:
            function ← MUX
    endcase
    case op of
        S8I:
            size ← 8
        S16LI, S16LAI, S16BI, S16BAI:
            size ← 16
        S32LI, S32LAI, S32BI, S32BAI:
            size ← 32
        S64LI, S64LAI, S64BI, S64BAI, SAAS64BAI, SAAS64LAI,
        SCAS64BAI, SCAS64LAI, SMAS64BAI, SMAS64LAI, SMUX64BAI, SMUX64LAI:
            size ← 64
        S128LI, S128LAI, S128BI, S128BAI:
            size ← 128
    endcase
    case op of
        S8I,
        S16LI, S16LAI, S16BI, S16BAI,
        S32LI, S32LAI, S32BI, S32BAI,
        S64LI, S64LAI, S64BI, S64BAI,
        SAAS64BAI, SAAS64LAI:
```


E 2 4

```
            rsize ← 64
        SCAS64BAI, SCAS64LAI, SMAS64BAI, SMAS64LAI, SMUX64BAI, SMUX64LAI:
            rsize ← 128
        S128LI, S128LAI, S128BI, S128BAI:
            rsize ← 128
    endcase
    case op of
        S8I:
            align ← undefined
        S16LI, S32LI, S64LI, S128LI,
        S16BI, S32BI, S64BI, S128BI:
            align ← false
        S16LAI, S32LAI, S64LAI, S128LAI,
        S16BAI, S32BAI, S64BAI, S128BAI,
        SAAS64BAI, SAAS64LAI, SCAS64BAI, SCAS64LAI,
        SMAS64BAI, SMAS64LAI, SMUX64BAI, SMUX64LAI:
            align ← true
    endcase
    case op of
        S8I:
            order ← undefined
        S16LI, S32LI, S64LI, S128LI,
        S16LAI, S32LAI, S64LAI, S128LAI,
        SAAS64LAI, SCAS64LAI, SMAS64LAI, SMUX64LAI:
            order ← L
        S16BI, S32BI, S64BI, S128BI,
        S16BAI, S32BAI, S64BAI, S128BAI,
        SAAS64BAI, SCAS64BAI, SMAS64BAI, SMUX64BAI:
            order ← B
    endcase
    a ← RegRead(ra, 64)
    VirtAddr ← a + (offset_11^50 || offset)
    if align then
        if (VirtAddr and ((size/8)-1)) ≠ 0 then
            raise AccessDisallowedByVirtualAddress
        endif
    endif
    m ← RegRead(rb, rsize)
    case function of
        NONE:
            StoreMemory(VirtAddr,size,order,m_{size-1..0})
        AAS:
            b ← LoadMemory(VirtAddr,size,order)
            StoreMemory(VirtAddr,size,order,m_{63..0}+b)
            RegWrite(rb, 64, b)
        CAS:
            b ← LoadMemory(VirtAddr,size,order)
            if (b = m_{63..0}) then
                StoreMemory(VirtAddr,size,order,m_{127..64})
            endif
            RegWrite(rb, 64, b)
        MAS:
            b ← LoadMemory(VirtAddr,size,order)
            n ← (m_{127..64} & m_{63..0}) | (b & ~m_{63..0})
            StoreMemory(VirtAddr,size,order,n)
            RegWrite(rb, 64, b)
        MUX:
```

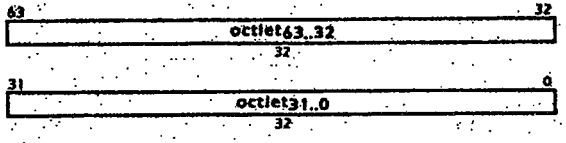
```
                b ← LoadMemory(VirtAddr,size,order)
                n ← (m127..64 & m63..0) | (b & ~m63..0)
                StoreMemory(VirtAddr,size,order,n)
    endcase
enddef
```

Exceptions

Reserved instruction
Access disallowed by virtual address
Access disallowed by tag
Access disallowed by global TLB
Access disallowed by local TLB
Access detail required by tag
Access detail required by local TLB
Access detail required by global TLB
Cache coherence intervention required by tag
Cache coherence intervention required by local TLB
Cache coherence intervention required by global TLB
Local TLB miss
Global TLB miss

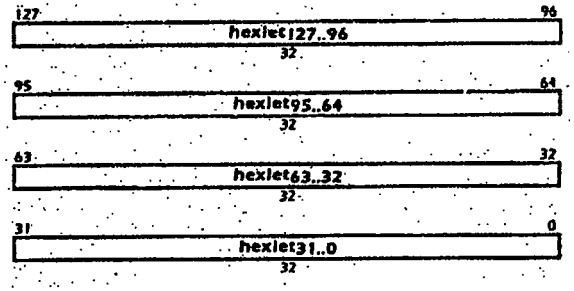
Octlet

An octlet is the catenation of 64 bits, and is the catenation of eight bytes:



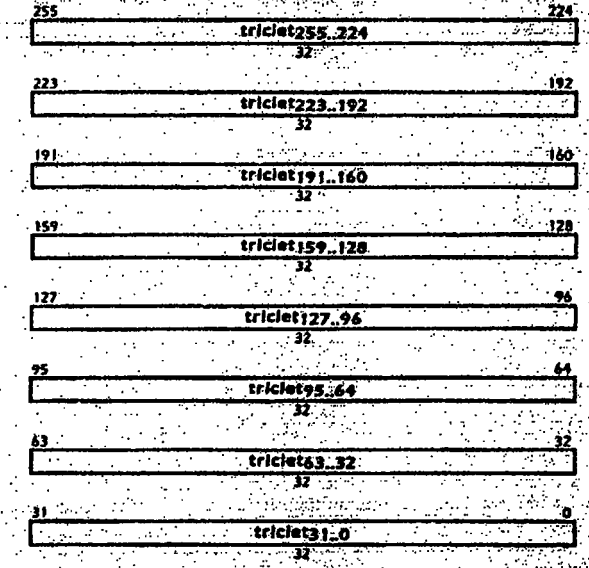
Hexlet

A hexlet is the catenation of 128 bits, and is the catenation of sixteen bytes:



Triclet

A triclet is the catenation of 256 bits, and is the catenation of thirty-two bytes:



Address

Zeus addresses, both virtual addresses and physical addresses, are octlet quantities.

Floating-point Data

Zeus's floating-point formats are designed to satisfy ANSI/IEEE standard 754-1985: Binary Floating-point Arithmetic. Standard 754 leaves certain aspects to the discretion of implementers: additional precision formats, encoding of quiet and signaling NaN values, details of production and propagation of quiet NaN values. These aspects are detailed below.

Zeus adds additional half-precision and quad-precision formats to standard 754's single-precision and double-precision formats. Zeus's double-precision satisfies standard 754's

precision requirements for a single-extended format, and Zeus's quad-precision satisfies standard 754's precision requirements for a double-extended format.

Each precision format employs fields labeled s (sign), e (exponent), and f (fraction) to encode values that are: (1) NaN: quiet and signaling, (2) infinities: (-1)^s∞, (3) normalized numbers: (-1)^s2^e-bias(1.f), (4) denormalized numbers: (-1)^s2^1-bias(0.f), and (5) zero: (-1)^s0.

Quiet NaN values are denoted by any sign bit value, an exponent field of all one bits, and a non-zero fraction with the most significant bit set. Quiet NaN values generated by default exception handling of standard operations have a zero sign bit, an exponent field of all one bits, a fraction field with the most significant bit set, and all other bits cleared.

Signaling NaN values are denoted by any sign bit value, an exponent field of all one bits, and a non-zero fraction with the most significant bit cleared.

Infinite values are denoted by any sign bit value, an exponent field of all one bits, and a zero fraction field.

Normalized number values are denoted by any sign bit value, an exponent field that is not all one bits or all zero bits, and any fraction field value. The numeric value encoded is (-1)^s2^e-bias(1.f). The bias is equal the value resulting from setting all but the most significant bit of the exponent field, half: 15, single: 127, double: 1023, and quad: 16383.

Denormalized number values are denoted by any sign bit value, an exponent field that is all zero bits, and a non-zero fraction field value. The numeric value encoded is (-1)^s2^1-bias(0.f).

Zero values are denoted by any sign bit value, and exponent field that is all zero bits, and a fraction field that is all zero bits. The numeric value encoded is (-1)^s0. The distinction between +0 and -0 is significant in some operations.

Half-precision Floating-point

Zeus half precision uses a format similar to standard 754's requirements, reduced to a 16-bit overall format. The format contains sufficient precision and exponent range to hold a 12-bit signed integer.

Single-precision Floating-point

Zeus single precision satisfies standard 754's requirements for "single."

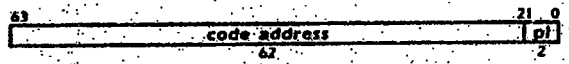
Double-precision Floating-point

Zeus double precision satisfies standard 754's requirements for "double."

Quad-precision Floating-point

Zeus quad precision satisfies standard 754's requirements for "double extended," but has additional fraction precision to use 128 bits.

The gateway contains two data items within its structure, a code address and a new privilege level:



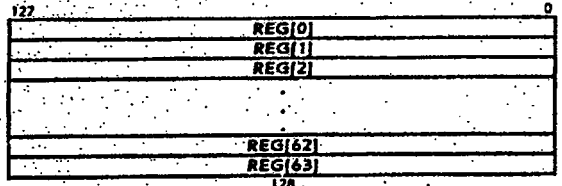
The virtual memory system can be used to designate a region of memory as containing gateways. Other data may be placed within the gateway region, provided that if an attempt is made to use the additional data as a gateway, that security cannot be violated. For example, 64-bit data or stack pointers which are aligned to at least 4 bytes and are in little-endian byte order have pl=0, so that the privilege level cannot be raised by attempting to use the additional data as a gateway.

User State

The user state consists of hardware data structures that are accessible to all conventional compiled code. The Zeus user state is designed to be as regular as possible, and consists only of the general registers, the program counter, and virtual memory. There are *no* specialized registers for condition codes, operating modes, rounding modes, integer multiply/divide, or floating-point values.

General Registers

Zeus user state includes 64 general registers. All are identical; there is no dedicated zero-valued register, and there are no dedicated floating-point registers.



Some Zeus instructions have 64-bit register operands. These operands are sign-extended to 128 bits when written to the register file, and the low-order 64 bits are chosen when read from the register file.

Definition

```
def val ← RegRead(m, size)
    case size of
        64:
            val ← RF[m]63..0
        128:
```

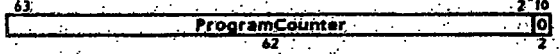


```
            val ← REG[m]
    endcase
enddef

def RegWrite(m, size, val)
    case size of
        64:
            REG[m] ← val63^64 || val63..0
        128:
            REG[m] ← val127..0
    endcase
enddef
```

Program Counter

The program counter contains the address of the currently executing instruction. This register is implicitly manipulated by branch instructions, and read by branch instructions that save a return address in a general register.

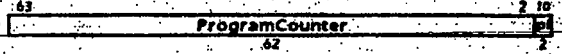


Privilege Level

The privilege level register contains the privilege level of the currently executing instruction. This register is implicitly manipulated by branch gateway and branch down instructions, and read by branch gateway instructions that save a return address in a general register.

Program Counter and Privilege Level

The program counter and privilege level may be packed into a single octlet. This combined data structure is saved by the Branch Gateway instruction and restored by the Branch Down instruction.



System state

The system state consists of the facilities not normally used by conventional compiled code. These facilities provide mechanisms to execute such code in a fully virtual environment. All system state is memory mapped, so that it can be manipulated by compiled code.

Fixed-point

Zeus provides load and store instructions to move data between memory and the registers, branch instructions to compare the contents of registers and to transfer control from one code address to another, and arithmetic operations to perform computation on the contents of registers, returning the result to registers.

Load and Store

The load and store instructions move data between memory and the registers. When loading data from memory into a register, values are zero-extended or sign-extended to fill the register. When storing data from a register into memory, values are truncated on the left to fit the specified memory region.

Load and store instructions that specify a memory region of more than one byte may use either little-endian or big-endian byte ordering; the size and ordering are explicitly specified in the instruction. Regions larger than one byte may be either aligned to addresses that are an even multiple of the size of the region or of unspecified alignment; alignment checking is also explicitly specified in the instruction.

Load and store instructions specify memory addresses as the sum of a base general register and the product of the size of the memory region and either an immediate value or another general register. Scaling maximizes the memory space which can be reached by immediate offsets from a single base general register, and assists in generating memory addresses within iterative loops. Alignment of the address can be reduced to checking the alignment of the first general register.

The load and store instructions are used for fixed-point data as well as floating-point and digital signal processing data; Zeus has a single bank of registers for all data types.

Swap instructions provide multithread and multiprocessor synchronization, using indivisible operations: add-swap, compare-swap, multiplex-swap, and double-compare-swap. A store-multiplex operation provides the ability to indivisibly write to a portion of an octlet. These instructions always operate on aligned octlet data, using either little-endian or big-endian byte ordering.

Branch

The fixed-point compare-and-branch instructions provide all arithmetic tests for equality and inequality of signed and unsigned fixed-point values. Tests are performed either between two operands contained in general registers, or on the bitwise and of two operands. Depending on the result of the compare, either a branch is taken, or not taken. A taken branch causes an immediate transfer of the program counter to the target of the branch, specified by a 12-bit signed offset from the location of the branch instruction. A non-taken branch causes no transfer; execution continues with the following instruction.

Other branch instructions provide for unconditional transfer of control to addresses too distant to be reached by a 12-bit offset, and to transfer to a target while placing the location

following the branch into a register. The branch through gateway instruction provides a secure means to access code at a higher privilege level, in a form similar to a normal procedure call.

Addressing Operations

A subset of general fixed-point arithmetic operations is available as addressing operations. These include add, subtract, Boolean, and simple shift operations. These addressing operations may be performed at a point in the Zeus processor pipeline so that they may be completed prior to or in conjunction with the execution of load and store operations in a "superpiping" pipeline in which other arithmetic operations are deferred until the completion of load and store operations.

Execution Operations

Many of the operations used for Digital Signal Processing (DSP), which are described in greater detail below, are also used for performing simple scalar operations. These operations perform arithmetic operations on values of 8-, 16-, 32-, 64-, or 128- bit sizes, which are right-aligned in registers. These execution operations include the add, subtract, boolean and simple shift operations which are also available as addressing operations, but further extend the available set to include three-operand add/subtract, three-operand boolean, dynamic shifts, and bit-field operations.

Floating-point

Zeus provides all the facilities mandated and recommended by ANSI/IEEE standard 754-1985: Binary Floating-point Arithmetic, with the use of supporting software.

Branch Conditionally

The floating-point compare-and-branch instructions provide all the comparison types required and suggested by the IEEE floating-point standard. These floating-point comparisons augment the usual types of numeric value comparisons with special handling for NaN (not-a-number) values. A NaN value compares as "unordered" with respect to any other value, even that of an identical NaN value.

Zeus floating-point compare-branch instructions do not generate an exception on comparisons involving quiet or signaling NaN values. If such exceptions are desired, they can be obtained by combining the use of a floating-point compare-set instruction, with either a floating-point compare-branch instruction on the floating-point operands or a fixed-point compare-branch on the set result.

Because the less and greater relations are anti-commutative, one of each relation that differs from another only by the replacement of an L with a G in the code can be removed by reversing the order of the operands and using the other code. Thus, an L relation can be used in place of a G relation by swapping the operands to the compare-branch or compare-set instruction.

No instructions are provided that branch when the values are unordered. To accomplish such an operation, use the reverse condition to branch over an immediately following unconditional branch, or in the case of an if-then-else clause, reverse the clauses and use the reverse condition.

The R relation can be used to determine the unordered condition of a single operand by comparing the operand with itself.

The following floating-point compare-branch relations are provided as instructions:

Mnemonic		Branch taken if values compare as:				Exception if	
code	C-like	Unord-ered	Greater	Less	Equal	unord-ered	invalid
E	==	F	F	F	T	no	no
LG	<>	F	T	T	F	no	no
L	<	F	F	T	F	no	no
GE	>=	F	T	F	T	no	no

compare-branch relations

Compare-set

The compare-set floating-point instructions provide all the comparison types supported as branch instructions. Zeus compare-set floating-point instructions may optionally generate an exception on comparisons involving quiet or signaling NaNs.

The following floating-point compare-set relations are provided as instructions:

Mnemonic		Result if values compare as:				Exception if	
code	C-like	Unord-ered	Greater	Less	Equal	unord-ered	invalid
E	==	F	F	F	T	no	no
LG	<>	F	T	T	F	no	no
L	<	F	F	T	F	no	no
GE	>=	F	T	F	T	no	no
EX	=?	F	F	F	T	no	yes
LGX	<>?	F	T	T	F	no	yes
LX	<?	F	F	T	F	yes	yes
GEX	>=?	F	T	F	T	yes	yes

compare-set relations

Arithmetic Operations

The basic operations supported in hardware are floating-point add, subtract, multiply, divide, square root and conversions among floating-point formats and between floating-point and binary integer formats.

Software libraries provide other operations required by the ANSI/IEEE floating-point standard.

The operations explicitly specify the precision of the operation, and round the result (or check that the result is exact) to the specified precision at the conclusion of each operation. Each of the basic operations splits operand registers into symbols of the specified precision and performs the same operation on corresponding symbols.

In addition to the basic operations, Zeus performs a variety of operations in which one or more products are summed to each other and/or to an additional operand. The instructions include a fused multiply-add (F.MUL.ADD.F), convolve (F.CON.F), matrix multiply (F.MUL.MAT.F), and scale-add (F.SCAL.ADD.F).

The results of these operations are computed as if the multiplies are performed to infinite precision, added as if in infinite precision, then rounded only once. Consequently, these operations perform these operations with no rounding of intermediate results that would have limited the accuracy of the result.

Rounding and exceptions

Rounding is specified within the instructions explicitly, to avoid explicit state registers for a rounding mode. Similarly, the instructions explicitly specify how standard exceptions (invalid operation, division by zero, overflow, underflow and inexact) are to be handled.[1]

When no rounding is explicitly named by the instruction (default), round to nearest rounding is performed, and all floating-point exception signals cause the standard-specified default result, rather than a trap. When rounding is explicitly named by the instruction (N: nearest, Z: zero, F: floor, C: ceiling), the specified rounding is performed, and floating-point exception signals other than inexact cause a floating-point exception trap. When X (exact, or exception) is specified, all floating-point exception signals cause a floating-point exception trap, including inexact.

This technique assists the Zeus processor in executing floating-point operations with greater parallelism. When default rounding and exception handling control is specified in floating-point instructions, Zeus may safely retire instructions following them, as they are guaranteed not to cause data-dependent exceptions. Similarly, floating-point instructions with N, Z, F, or C control can be guaranteed not to cause data-dependent exceptions once the operands have been examined to rule out invalid operations, division by zero, overflow or underflow exceptions. Only floating-point instructions with X control, or when exceptions cannot be ruled out with N, Z, F, or C control need to avoid retiring following instructions until the final result is generated.

ANSI/IEEE standard 754-1985 specifies information to be given to trap handlers for the five floating-point exceptions. The Zeus architecture produces a precise exception. (The program counter points to the instruction that caused the exception and all register state is present) from which all the required information can be produced in software, as all source operand values and the specified operation are available.

[1] U.S. Patent 5,812,439 describes this "Technique of incorporating floating point information into processor instructions."

fetch unit of a branch-conditional instruction, so that instructions at the target of the branch can be fetched in advance of the branch-conditional instruction reaching the execution pipeline. Placing the branch hint as early as possible, and at a point where the extra instruction will not reduce the execution rate optimizes performance. In other words, an optimizing compiler should insert the branch-hint instruction as early as possible in the basic block where the parcel will contain at most one other "front-end" instruction.

Additional Load and Execute Resources

Studies of the dynamic distribution of Zeus instructions on various benchmark suites indicate that the most frequently-issued instruction classes are load instructions and execute instructions. In a high-performance Zeus implementation, it is advantageous to consider execution pipelines in which the ability to target the machine resources toward issuing load and execute instructions is increased.

One of the means to increase the ability to issue execute-class instructions is to provide the means to issue two execute instructions in a single-issue string. The execution unit actually requires several distinct resources, so by partitioning these resources, the issue capability can be increased without increasing the number of functional units, other than the increased register file read and write ports. The partitioning favored for the initial implementation places all instructions that involve shifting and shuffling in one execution unit, and all instructions that involve multiplication, including fixed-point and floating-point multiply and add in another unit. Resources used for implementing add, subtract, and bitwise logical operations may be duplicated, being modest in size compared to the shift and multiply units, or shared between the two units, as the operations have low-enough latency that two operations might be pipelined within a single issue cycle. These instructions must generally be independent, except perhaps that two simple add, subtract, or bitwise logical instructions may be performed dependently, if the resources for executing simple instructions are shared between the execution units.

One of the means to increase the ability to issue load-class instructions is to provide the means to issue two load instructions in a single-issue string. This would generally increase the resources required of the data fetch unit and the data cache, but a compensating solution is to steal the resources for the store instruction to execute the second load instruction. Thus, a single-issue string can then contain either two load instructions, or one load instruction and one store instruction, which uses the same register read ports and address computation resources as the basic 3-instruction string. This capability also may be employed to provide support for unaligned load and store instructions, where a single-issue string may contain as an alternative a single unaligned load or store instruction which uses the resources of the two load-class units in concert to accomplish the unaligned memory operation.

Result Forwarding

When temporally adjacent instructions are executed by separate resources, the results of the first instruction must generally be forwarded directly to the resource used to execute the second instruction, where the result replaces a value which may have been fetched from a register file. Such forwarding paths use significant resources. A Zeus implementation must

generally provide forwarding resources so that dependencies from earlier instructions within a string are immediately forwarded to later instructions, except between a first and second execution instruction as described above. In addition, when forwarding results from the execution units back to the data fetch unit, additional delay may be incurred.

Store

These operations add the contents of two registers to produce a virtual address, and store the contents of a register into memory.

Operation codes

S.8[2]	Store byte
S.16.B	Store double big-endian
S.16.AB	Store double aligned big-endian
S.16.L	Store double little-endian
S.16.AL	Store double aligned little-endian
S.32.B	Store quadlet big-endian
S.32.AB	Store quadlet aligned big-endian
S.32.L	Store quadlet little-endian
S.32.AL	Store quadlet aligned little-endian
S.64.B	Store octlet big-endian
S.64.AB	Store octlet aligned big-endian
S.64.L	Store octlet little-endian
S.64.AL	Store octlet aligned little-endian
S.128.B	Store hexlet big-endian
S.128.AB	Store hexlet aligned big-endian
S.128.L	Store hexlet little-endian
S.128.AL	Store hexlet aligned little-endian
S.MUX64.AB	Store multiplex octlet aligned big-endian
S.MUX64.AL	Store multiplex octlet aligned little-endian

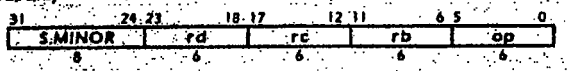
Selection

number format	op	size	alignment	ordering
byte		8		
integer		16 32 64 128		L B
integer aligned		16 32 64 128	A	L B
multiplex	MUX	64	A	L B

Format

op rd,rc,rb

op(rd,rc,rb)



[2] S.8 need not specify byte ordering, nor need it specify alignment checking, as it stores a single byte.

Description

An operand size, expressed in bytes, is specified by the instruction. A virtual address is computed from the sum of the contents of register rc and the contents of register rb multiplied by operand size. The contents of register rd, treated as the size specified, is stored in memory using the specified byte order.

If alignment is specified, the computed virtual address must be aligned, that is, it must be an exact multiple of the size expressed in bytes. If the address is not aligned an "access disallowed by virtual address" exception occurs.

Definition

```
def StoreOp(rd,rc,rb) as
    case op of
        S8:
            size <- 8
        S16L, S16AL, S16B, S16AB:
            size <- 16
        S32L, S32AL, S32B, S32AB:
            size <- 32
        S64L, S64AL, S64B, S64AB,
        SMUX64AB, SMUX64AL:
            size <- 64
        S128L, S128AL, S128B, S128AB:
            size <- 128
    endcase
    lsize <- log(size)
    case op of
        S8:
            order <- undefined
        S16L, S32L, S64L, S128L,
        S16AL, S32AL, S64AL, S128AL, SMUX64ALt:
            order <- L
        S16B, S32B, S64B, S128B,
        S16AB, S32AB, S64AB, S128AB, SMUX64ABt:
            order <- B
    endcase
    c <- RegRead(rc, 64)
    b <- RegRead(rb, 64)
    VirtAddr <- c + (b[64-lsize:0] || 0[lsize-3])
    case op of
        S16AL, S32AL, S64AL, S128AL,
        S16AB, S32AB, S64AB, S128AB,
        SMUX64AB, SMUX64AL:
            if (lsize-1:0 ≠ 0) then
                raise AccessDisallowedByVirtualAddress
            endif
        S16L, S32L, S64L, S128L,
        S16B, S32B, S64B, S128B:
            S8:
    endcase
    d <- RegRead(rd, 128)
    case op of
```

Store Double compare Swap

These operations compare two 64-bit values in a register against two 64-bit values stored from two 64-bit memory locations, as specified by two 64-bit addresses in a register, and if equal, store two new 64-bit values from a register into the memory locations. The values read from memory are compared and placed in a register.

Operation codes

SDCS64AB	Store double compare swap octlet aligned big-endian
SDCS64AL	Store double compare swap octlet aligned little-endian

Format

op rd,rc,rb

rdesp(rd,rc,rb)

Description

Two virtual addresses are extracted from the low order bits of the contents of registers rc and rb. Two 64-bit comparison values are extracted from the high order bits of the contents of registers rc and rb. Two 64-bit replacement values are extracted from the contents of register rd. The contents of memory using the specified byte order are read from the specified addresses, treated as 64-bit values, compared against the specified comparison values, and if both read values are equal to the comparison values, the two replacement values are written to memory using the specified byte order. If either are unequal, no values are written to memory. The loaded values are canonized and placed in the register specified by rd.

The virtual addresses must be aligned, that is, it must be an exact multiple of the size expressed in bytes. If the address is not aligned an "access disallowed by virtual address" exception occurs.

Definition

```
def StoreDoubleCompareSwap(op,rd,rc,rb) as
    size ← 64
    base ← bytes(size)
    addr0 ← GPR(rc)<63:0>
    addr1 ← GPR(rc)<127:64>
    cmp0 ← GPR(rb)<63:0>
    cmp1 ← GPR(rb)<127:64>
    rep0 ← GPR(rd)<63:0>
    rep1 ← GPR(rd)<127:64>
    order ← L
    SDCS64AB:
        order ← B
    SDCS64AL:
        if addr0<2:0> ≠ 0 or addr1<2:0> ≠ 0 then
            SignalException(AccessDisallowedByVirtualAddress)
        if addr0 ≠ addr1 + base then
            SignalException(AccessDisallowedByVirtualAddress)
        if addr0 = 0 then
            SignalException(AccessDisallowedByVirtualAddress)
        if addr1 = 0 then
            SignalException(AccessDisallowedByVirtualAddress)
        if addr0 = addr1 then
            SignalException(AccessDisallowedByVirtualAddress)
        if addr0 = addr1 - base then
            SignalException(AccessDisallowedByVirtualAddress)
        if addr0 = addr1 + base
```

```
endcase
c ← RegRead(rc, 128)
b ← RegRead(rb, 128)
d ← RegRead(rd, 128)
if b2..0 = 0 or b2..0 = 4 then
  raise AccessDisallowedByVirtualAddress
case
  a ← LoadMemory(MR3,0,64,b,64,order) || LoadMemory(MR3,0,64,b+8,64,order)
  if IR<127:64> != 0 then
    StoreMemory(MR3,0,64,b,64,order,d<127:64>
    StoreMemory(MR3,0,64,b+8,64,order,d<63:0>
  else
    StoreMemory(MR3,0,64,b,64,order,d<63:0>
endcase
RegWrite(rd, 128, a)
endcase
```

Exceptions

Access disallowed by virtual address
Access disallowed by seg
Access disallowed by global TB
Access disallowed by local TB
Access disallowed supplied by seg
Access disallowed supplied by local TB
Access disallowed supplied by global TB
Local TB miss
Global TB miss

Store Immediate

These operations add the contents of a register to a sign-extended immediate value to produce a virtual address, and store the contents of a register into memory.

Operation codes

$LOpst	
$L16AB	Store immediate byte
$L16AB	Store immediate double aligned big-endian
$L16.B	Store immediate double big-endian
$L16AL	Store immediate double aligned little-endian
$L16E	Store immediate double little-endian
$L32AB	Store immediate quadlet aligned big-endian
$L32.B	Store immediate quadlet big-endian
$L32AL	Store immediate quadlet aligned little-endian
$L32L	Store immediate quadlet little-endian
$L64AB	Store immediate octlet aligned big-endian
$L64.B	Store immediate octlet big-endian
$L64AL	Store immediate octlet aligned little-endian
$L64L	Store immediate octlet little-endian
$L128AB	Store immediate hextet aligned big-endian
$L128.B	Store immediate hextet big-endian
$L128AL	Store immediate hextet aligned little-endian
$L128L	Store immediate hextet little-endian
$MSKL64AB	Store masked immediate octlet aligned big-endian
$MSKL64AL	Store masked immediate octlet aligned little-endian

Selection

number format	op	size	alignment	ordering
byte	8			
integer		16 32 64 128		L B
integer aligned		16 32 64 128	A	L B
multiples	MUX	64	A	L B

Format

$op.size.align.order rd,rc,offset

sop{size}{align}order{rd,rc,offset}

[2]8 need not specify byte ordering, nor need it specify alignment checking, as it stores a single byte.

Description

An operand size, expressed in bytes, is specified by the instruction. A virtual address is computed from the sum of the contents of register rc and the sign-extended value of the offset field, multiplied by the operand size. The contents of register rd, treated as the size specified, are written to memory using the specified byte order.

The computed virtual address must be aligned, that is, it must be an exact multiple of the size expressed in bytes. If the address is not aligned an "access disallowed by virtual address" exception occurs.

Definition

```
def StoreImmediate(rd,rc,offset) as
    case op of
        [0b0]
            disp <- 8
            S1[4], S1[4A], S1[6B], S1[6AB]
        [0b1]
            disp <- 16
            S32[], S32[A], S32[B], S32[AB]
        [0b2]
            disp <- 32
            S64[], S64[A], S64[B], S64[AB], SMU[X64[AB], SMU[X64[A],
        [0b3]
            disp <- 64
            S128[], S128[A], S128[B], S128[AB],
        endcase
        disp <- 128
    endcase
    Here <- bigend
    case op of
        [0b0]
            order <- undefined
            S1[4], S32[], S64[], S128[],
            S1[4], S32[A], S64[A], S128[A], SMU[X64[A],
            order <- 1
            S1[6B], S32[B], S64[B], S128[B],
            S1[6AB], S32[AB], S64[AB], S128[AB], SMU[X64[AB],
            order <- 8
        endcase
        c <- RegRead(rc, 64)
        VAddr <- c + (offset || offset || 0b0^3)
        case op of
            S1[4], S32[], S64[], S128[],
            S1[4A], S32[A], S64[A], S128[A],
            S1[6B], S32[B], S64[B], S128[B],
            S1[6AB], S32[AB], S64[AB], S128[AB],
            SMU[X64[], SMU[X64[A],
            if Fault <> 0 then
                raise AccessDisallowedByVirtualAddress
            endif
            S1[4], S32[], S64[], S128[],
            S1[6B], S32[B], S64[B], S128[B]:
                order <- RegRead(rd, 128)
        endcase
        case op of
            [0]
                S1[4], S1[4A], S1[6B], S1[6AB],
                S32[], S32[A], S32[B], S32[AB],
```


Description

S64[], S64[A], S64[B], S64[AB],
S128[], S128[A], S128[B], S128[AB]
SMU[X64[], SMU[X64[A],
[0b1]
d <- LoadMemory(VAddr, order, disp)
m <- [0]^127 || 0b1
StoreMemory(VAddr,order,d,m)
endcase

Exceptions

Access disallowed by virtual address
Access disallowed by page TB
Access disallowed by local TB
Access disallowed by global TB
Access disallowed by page TB
Access disallowed by local TB
Access disallowed by global TB
Local TB miss
Global TB miss

```
case op of
    E.MUL.ADD.X.L, E.MUL.ADD.X.L.U, E.MUL.ADD.X.L.M:
        h ← 2*size + 1
    E.MUL.ADD.X.L.C:
        h ← (2*size) + 2
endcase
r ← h - size - sh + 1 - (cs and bs)
for i ← 0 to 128-size by size
    di ← {ds and d(h-size-1)}^(h-size-r) || {d(h-size-1..i) || 0^i}
    case op of
        E.MUL.ADD.X.L, E.MUL.ADD.X.L.M, E.MUL.ADD.X.L.U,
            p ← mul(size,h,cs,c,i,bs,b,i) + di
        E.MUL.ADD.X.L.C:
            if i & size = 0 then
                p ← mul(size,h,cs,c,i,bs,b,i) - mul(size,h,cs,c,i+size,ps,b,i+size) + di
            else
                p ← mul(size,h,cs,c,i,bs,b,i+size) + mul(size,h,cs,c,i,bs,b,i+size) + di
            endif
    endcase
    case rnd of
        none, N:
            s ← 0^(h-r) || ~p_r || p_r^(r-1)
        Z:
            s ← 0^(h-r) || p_(h-1)
        F:
            s ← 0^h
        C:
            s ← 0^(h-r) || 1 || 1^r
    endcase
    v ← {ds & p_(h-1)}||1||p|| {011}
    if (v_(h-1+size) = {ds & v_(r+size-1)}^(h-1+r-size)) then
            a_(size-1+i..i) ← v_(size-1+r..r)
    else
            a_(size-1+i..i) ← ds ? (v_h || ~v_(h-1)^(size-1)) : 1^size
    endif
endfor
RegWrite(rd, 128, a)
enddef
```

Exceptions

ReservedInstruction

Ensemble Floating-point

These operations take two values from registers, perform a group of floating-point arithmetic operations on partitions of bits in the operands, and place the catenated results in a register.

Operation codes

E.ADD.F.16	Ensemble add floating-point half
E.ADD.F.16.C	Ensemble add floating-point half ceiling
E.ADD.F.16.F	Ensemble add floating-point half floor
E.ADD.F.16.N	Ensemble add floating-point half nearest
E.ADD.F.16.X	Ensemble add floating-point half exact
E.ADD.F.16.Z	Ensemble add floating-point half zero
E.ADD.F.32	Ensemble add floating-point single
E.ADD.F.32.C	Ensemble add floating-point single ceiling
E.ADD.F.32.F	Ensemble add floating-point single floor
E.ADD.F.32.N	Ensemble add floating-point single nearest
E.ADD.F.32.X	Ensemble add floating-point single exact
E.ADD.F.32.Z	Ensemble add floating-point single zero
E.ADD.F.64	Ensemble add floating-point double
E.ADD.F.64.C	Ensemble add floating-point double ceiling
E.ADD.F.64.F	Ensemble add floating-point double floor
E.ADD.F.64.N	Ensemble add floating-point double nearest
E.ADD.F.64.X	Ensemble add floating-point double exact
E.ADD.F.64.Z	Ensemble add floating-point double zero
E.ADD.F.128	Ensemble add floating-point quad
E.ADD.F.128.C	Ensemble add floating-point quad ceiling
E.ADD.F.128.F	Ensemble add floating-point quad floor
E.ADD.F.128.N	Ensemble add floating-point quad nearest
E.ADD.F.128.X	Ensemble add floating-point quad exact
E.ADD.F.128.Z	Ensemble add floating-point quad zero
E.DIV.F.16	Ensemble divide floating-point half
E.DIV.F.16.C	Ensemble divide floating-point half ceiling
E.DIV.F.16.F	Ensemble divide floating-point half floor
E.DIV.F.16.N	Ensemble divide floating-point half nearest
E.DIV.F.16.X	Ensemble divide floating-point half exact
E.DIV.F.16.Z	Ensemble divide floating-point half zero
E.DIV.F.32	Ensemble divide floating-point single
E.DIV.F.32.C	Ensemble divide floating-point single ceiling
E.DIV.F.32.F	Ensemble divide floating-point single floor
E.DIV.F.32.N	Ensemble divide floating-point single nearest
E.DIV.F.32.X	Ensemble divide floating-point single exact
E.DIV.F.32.Z	Ensemble divide floating-point single zero
E.DIV.F.64	Ensemble divide floating-point double
E.DIV.F.64.C	Ensemble divide floating-point double ceiling

E.DIV.F.64.F	Ensemble divide floating-point double floor
E.DIV.F.64.N	Ensemble divide floating-point double nearest
E.DIV.F.64.X	Ensemble divide floating-point double exact
E.DIV.F.64.Z	Ensemble divide floating-point double zero
E.DIV.F.128	Ensemble divide floating-point quad
E.DIV.F.128.C	Ensemble divide floating-point quad ceiling
E.DIV.F.128.F	Ensemble divide floating-point quad floor
E.DIV.F.128.N	Ensemble divide floating-point quad nearest
E.DIV.F.128.X	Ensemble divide floating-point quad exact
E.DIV.F.128.Z	Ensemble divide floating-point quad zero
E.MUL.C.F.16	Ensemble multiply complex floating-point half
E.MUL.C.F.32	Ensemble multiply complex floating-point single
E.MUL.C.F.64	Ensemble multiply complex floating-point double
E.MUL.F.16	Ensemble multiply floating-point half
E.MUL.F.16.C	Ensemble multiply floating-point half ceiling
E.MUL.F.16.F	Ensemble multiply floating-point half floor
E.MUL.F.16.N	Ensemble multiply floating-point half nearest
E.MUL.F.16.X	Ensemble multiply floating-point half exact
E.MUL.F.16.Z	Ensemble multiply floating-point half zero
E.MUL.F.32	Ensemble multiply floating-point single
E.MUL.F.32.C	Ensemble multiply floating-point single ceiling
E.MUL.F.32.F	Ensemble multiply floating-point single floor
E.MUL.F.32.N	Ensemble multiply floating-point single nearest
E.MUL.F.32.X	Ensemble multiply floating-point single exact
E.MUL.F.32.Z	Ensemble multiply floating-point single zero
E.MUL.F.64	Ensemble multiply floating-point double
E.MUL.F.64.C	Ensemble multiply floating-point double ceiling
E.MUL.F.64.F	Ensemble multiply floating-point double floor
E.MUL.F.64.N	Ensemble multiply floating-point double nearest
E.MUL.F.64.X	Ensemble multiply floating-point double exact
E.MUL.F.64.Z	Ensemble multiply floating-point double zero
E.MUL.F.128	Ensemble multiply floating-point quad
E.MUL.F.128.C	Ensemble multiply floating-point quad ceiling
E.MUL.F.128.F	Ensemble multiply floating-point quad floor
E.MUL.F.128.N	Ensemble multiply floating-point quad nearest
E.MUL.F.128.X	Ensemble multiply floating-point quad exact
E.MUL.F.128.Z	Ensemble multiply floating-point quad zero

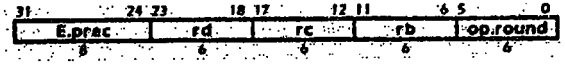
Selection

class	op	prec				round/trap
add	EADDF	16	32	64	128	none C F N X Z
divide	EDIVF	16	32	64	128	none C F N X Z
multiply	EMULF	16	32	64	128	none C F N X Z
complex multiply	EMULCF	16	32	64		none

Format

E.op.prec.round rd=rc,rb

rd=eopprecround(rc,rb)



Description

The contents of registers ra and rb are combined using the specified floating-point operation. The result is placed in register rc. The operation is rounded using the specified rounding option or using round-to-nearest if not specified. If a rounding option is specified, the operation raises a floating-point exception if a floating-point invalid operation, divide by zero, overflow, or underflow occurs, or when specified, if the result is inexact. If a rounding option is not specified, floating-point exceptions are not raised, and are handled according to the default rules of IEEE 754.

Definition

```
def mul(size,v,i,w,j) as
    mul ← fmul(F(size,v[size-1+i,i]),F(size,w[size-1+j,j]))
enddef

def EnsembleFloatingPoint(op,prec,round,ra,rb,rc) as
    c ← RegRead(rc, 128)
    b ← RegRead(rb, 128)
    for i ← 0 to 128-prec by prec
        ci ← F(prec,c[i+prec-1,i])
        bi ← F(prec,b[i+prec-1,i])
        case op of
            EADD.F:
                ai ← fadd(ci,bi,round)
            EMUL.F:
                ai ← fmul(ci,bi)
            EMUL.C.F:
                if i and prec[ then
                    ai ← fadd(mul(prec,c,i,b,i-prec[, mul(prec,c,i-prec,b,i))
                else
                    ai ← fsub(mul(prec,c,i,b,i[, mul(prec,c,i+prec,b,i+prec))
                endif
            EDIV.F:
                ai ← fdiv(ci,bi)
        endcase
        a[i+prec-1:i] ← PackF(prec, ai, round)
    endfor
    RegWrite(rd, 128, a)
enddef
```

Exceptions

floating-point arithmetic